User's Guide to JADAMILU

FORTRAN routines to solve symmetric/Hermitian (generalized) eigenvalue problems

Matthias Bollhöfer [1,*] and Yvan Notay [2,*]

[1] *Institut "Computational Mathematics", Technische Universität Braunschweig, D–38106 Braunschweig, Germany.*

[2] *Service de Métrologie Nucléaire, Université Libre de Bruxelles (C.P. 165/84), 50, Av. F.D. Roosevelt, B-1050 Brussels, Belgium.*

October 2006
Revised December 2008

Abstract

This manual gives an introduction to the use of JADAMILU, a set of Fortran 77 routines to compute selected eigenvalues and associated eigenvectors of large scale real symmetric or complex Hermitian matrices. Generalized eigenvalue problems with positive definite mass matrix can also be solved. The eigenvalues sought can either be the smallest ones or those closest to some specified target inside the spectrum. The package is based on the Jacobi-Davidson method. Key features are modest memory requirements, robust convergence to accurate solutions, and the automatic setup of an in-built preconditioner. On the other hand, the program is flexible and easy to use by non-experts. If the matrix is sparse and explicitly available, the user can simply pass it to the program and let JADAMILU compute the desired eigenvalues. If not, then a crude sparse approximation which may even be diagonal, can be passed. In the latter case the user only needs to define a routine that performs matrix-vector products.

[*]Supported by the "Fonds National de la Recherche Scientifique", Maître de recherches.

Key words. eigensolver, Jacobi–Davidson, preconditioning, software

AMS subject classification. 65F10

Contents

1 Introduction

JADAMILU allows to compute selected solutions of standard

$$A\,\mathbf{u} \;=\; \lambda\,\mathbf{u}$$

or generalized

$$A\,\mathbf{u} \;=\; \lambda\,B\,\mathbf{u}$$

eigenvalue problem, where A, B are large sparse symmetric or complex Hermitian matrices, with B being positive definite in case of a generalized eigenvalue problem.

JADAMILU is an inverse free software: neither the inverse of (shifted) A nor that of B is used even implicitly. The algorithms used are based on the Jacobi–Davidson method ([13, 14]), one of the state-of-the-art methods for computing a few selected eigenvalues and associated eigenvectors of large sparse matrices. Roughly spoken, Jacobi-Davidson can be viewed as a Newton-like method for computing an eigenvalue and its associated eigenvector. Similar to Newton's method where one has to solve a system with the Jacobian at every step, in the Jacobi–Davidson method one has to solve a correction equation with the shifted input matrix $A - sI$ or $A - sB$ (plus some projection). This system needs not be solved accurately, and, actually, they are solved inside JADAMILU using the conjugate gradient method with state-of-the-art preconditioning [5] based on integrated multilevel incomplete LU (ILU) factorization of the shifted matrix ($A - sI$ or $A - sB$).

1.1 How to use this guide

This guide is self contained, but does not describe methods and algorithms used in JADAMILU, for which we refer to [4]. The latter paper also describes some real life applications. We strongly recommend its reading for an enlightened use of the package.

On the other hand, main driver routines are reproduced in the Appendix with the comment lines describing precisely each parameter. It is a good idea to have a look at the listing of a driver routine while reading a section of this guide describing its usage.

1.2 Main driver routines and versions

JADAMILU has three main driver routines, `PJD`, `PJDREVCOM` and `JDREVCOM`, which are declined in different versions. Each driver is available in double precision (prefix `D`), double complex (prefix `Z`), single precision (prefix `S`) and single complex (prefix `C`) arithmetic. Each driver has also a variant for standard eigenvalue problems (no suffix) and a variant for generalized eigenvalue problems (suffix `_GEP`). In addition, for each arithmetic, JADAMILU provides a routine `PJDCLEANUP` that releases internal memory (same routine for both generalized and non generalized cases).

Therefore available driver routines are

DPJD	ZPJD	SPJD	CPJD
DPJDREVCOM	ZPJDREVCOM	SPJDREVCOM	CPJDREVCOM
DJDREVCOM	ZJDREVCOM	SJDREVCOM	CJDREVCOM
DPJD_GEP	ZPJD_GEP	SPJD_GEP	CPJD_GEP
DPJDREVCOM_GEP	ZPJDREVCOM_GEP	SPJDREVCOM_GEP	CPJDREVCOM_GEP
DJDREVCOM_GEP	ZJDREVCOM_GEP	SJDREVCOM_GEP	CJDREVCOM_GEP
DPJDCLAENUP	ZPJDCLAENUP	SPJDCLAENUP	CPJDCLAENUP

The usage of these driver routines is pretty much independent of the version (for standard or generalized eigenvalue problem) and of the arithmetic. Therefore, most of this guide (Sections 2 to 7) describes, by way of example, the usage of the double precision routines for standard eigenproblems `DPJD`, `DPJDREVCOM` `DJDREVCOM` and `DPJDCLAENUP`. Peculiarities related to generalized eigenvalue problems and to complex arithmetic (for complex Hermitian matrices) are discussed in Sections 8 and 9, respectively.

In the Appendix, we only reproduce driver routines for double precision, because the argument description is the same for all arithmetics, besides of course the argument type of non integer variables. At any rate, the source files of all drivers are provided with the package, see the README file for details.

Note that we do not recommend single precision versions, which are provided for the sake of completeness and should be considered with care. As you may see by running the examples provided with the package, accuracy is limited. Further, for ill-conditioned problems, rounding errors may spoil any accuracy and/or convergence. If saving memory is important and/or if single precision arithmetic is much faster on your machine, we recommend to use the double precision version but with preconditioner in single precision, see the description of parameter `ICNTL(5)` in Section 4.3.

1.3 Which driver use

`PJD` is the major driver of this package and we recommend its use, which is documented in Sections 2 to 7 but Sections 3.4, 4.4 and 4.5. `PJDREVCOM` has to be used when the matrix is not explicitly available or when passing it to JADAMILU would require excessive memory; its usage is more particularly documented in Sections 3.4 and 4.5, but remarks on `PJD` in other sections also apply to `PJDREVCOM`. Eventually, you should use `JDREVCOM` if you want to use your own preconditioner instead of the one provided by JADAMILU. Its usage is documented in Section 4.4; remarks on `PJD`/`PJDREVCOM` also apply, except of course those related to preconditioning, mainly in Section 4.

1.4 Release and backward compatibility

This guide describes JADAMILU 2.0 and above. From previous release (the first version, without revision number) the naming of the routines has been changed and some control parameter has been added. We nevertheless provide full backward compatibility, i.e. application programs based on this previous release need not be modified.

If you want nevertheless upgrade a program working with previous release, you just need to add a `D` as prefix to the driver called, and extend the declared length of control parameter `ICNTL` from 4 to 5, while initializing `ICNTL(5)=0`.

1.5 Disclaimer and copyright

THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK.

Using JADAMILU is free for non commercial applications (For commercial use, please contact the authors). You can acknowledge, using reference [4], the contribution of this package in any scientific publication dependent upon the use of the package. You shall use reasonable endeavors to notify the authors of the package of this publication.

JADAMILU also uses some external packages. Please check Appendix B for additional copyright notice.

1.6 Running the examples

Source files for all listings in this guide are provided with the package. Therefore, you can run the examples on your own. Please note that some decisions taken inside JADAMILU, like recomputing or not a preconditioner, are based on timing measures that are machine dependant. Therefore, RUNNING THE EXAMPLES ON DIFFERENT MACHINES MAY PRODUCE SLIGHTLY DIFFERENT OUTPUT. This is harmless as long as the computed eigenvalues are the same up to the required accuracy.

In all cases, we give the source files for standard eigenvalue problems in double precision arithmetic, but some source files are also provided for other arithmetics and/or generalized eigenvalue problems. They are not discussed in this guide, but you can run them on your own, and compare the source code with the one of the standard version; you will see that only few straightforward modifications are needed.

2 Getting started

Given your real symmetric matrix, suppose you plan to compute a few **eigenvalues closest to the origin** and their associated eigenvectors; JADAMILU asks you to do the following steps.

1. Provide the upper triangular part of your matrix in compressed sparse row format. Suppose your matrix is

$$A = \begin{pmatrix} 0 & 5 & & & \\ 5 & 1 & 5 & & \\ & 5 & 2 & 5 & \\ & & 5 & 3 & 5 \\ & & & 5 & 4 \end{pmatrix} \in \mathbb{R}^{5,5},$$

then the compressed sparse row format of the upper triangular part of A is given by the following three vectors.

$$\begin{aligned} \texttt{IA} &= \left[\begin{array}{cccccc} 1 & 3 & 5 & 7 & 9 & 10 \end{array}\right] \\ \texttt{JA} &= \left[\begin{array}{c|c|c|c|cc|c} 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 5 \end{array}\right] \\ \texttt{A} &= \left[\begin{array}{cc|cc|cc|cc|c} 0.0 & 5.0 & 1.0 & 5.0 & 2.0 & 5.0 & 3.0 & 5.0 & 4.0 \end{array}\right] \end{aligned}$$

The first array `IA` indicates where every row starts ("10" would be the start of row 6). `JA` carries the column indices of the upper triangular part, `A` the associated numerical values.

Note that even zero diagonal entries must be stored. For a more detailed description see Appendix B. Here is a sample code how one would enter the matrix in FORTRAN if it were a 1000×1000 matrix.

Listing 1: Sample code for compressed sparse row format

```
      INTEGER    N
      PARAMETER (N=1000)
      INTEGER IA(N+1), JA(2*N-1), I,K
      DOUBLE PRECISION A(2*N-1)
      K=1
      DO I=1,N
c        start of row I
         IA(I)=K
c        diagonal entry
         JA(K)=I
         A(K)=DBLE(I-1)
         K=K+1
c        entries above the diagonal part in row I
         IF (I .LT. N) THEN
            JA(K)=I+1
            A(K)=5.0D0
            K=K+1
         END IF
      END DO
c     final reference
      IA(N+1)=K
```

2. call PJD to compute the eigenvalues and eigenvectors. Here you have to decide how many eigenvalues you want, whether or not you have some initial guesses, how much memory you would like to spend, how to choose the drop tolerance, ...

   In the most simple case use the following sample code to compute the five (MAXEIG=5) smallest eigenvalues of your $N \times N$ matrix (say, size N=1000).

Listing 2: EXAMPLE1: Simple driver to compute inner eigenvalues

```
      INTEGER     N, MAXEIG, MAXSP
      PARAMETER (N=1000,MAXEIG=5,MAXSP=20)
      INTEGER     LX
      PARAMETER (LX=N*(3*MAXSP+MAXEIG+1)+4*MAXSP*MAXSP)
      DOUBLE PRECISION   EIGS(MAXEIG), RES(MAXEIG), X(LX)
      INTEGER            NEIG, MADSPACE, ISEARCH, NINIT, ICNTL(5)
      INTEGER            ITER, IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, SHIFT, GAP, MEM, DROPTOL
c
c     ...
c     declare IA, JA and A and initialize matrix
c     ...
c
c     standard diagnostic report on standard output
      IPRINT=6
c     we want the eigenvalues near some target sigma
      ISEARCH=2
c     target, where are we seeking for eigenvalues. Here: near origin
      SIGMA=0.0d0
c     elbow space factor for the fill computed during the ILU
      MEM=20.0
c     tolerance for discarded fill
      DROPTOL=1.d-3
c     number of wanted eigenvalues
      NEIG=MAXEIG
c     no initial approximate eigenvectors
      NINIT=0
c     desired size of the search space
      MADSPACE=MAXSP
c     maximum number of iteration steps
      ITER=1000
c     tolerance for the eigenvector residual
      TOL=1.0d-10
c     additional parameters set to default
      ICNTL(1)=0
      ICNTL(2)=0
      ICNTL(3)=0
      ICNTL(4)=0
      ICNTL(5)=0
c
c     compute NEIG eigenvalues near SIGMA
      CALL DPJD( N,A,JA,IA, EIGS, RES, X, LX, NEIG,
     +           SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +           SHIFT, DROPTOL, MEM, ICNTL, IPRINT, INFO, GAP)
c
c     release internal memory
      CALL DPJDCLEANUP
```

If completed successfully, PJD will return the desired eigenvalues in `EIGS(1)`,..., `EIGS(5)`. `X(1+N*(i-1):N*i)`, for `i=1`,...,5 contains the associated normalized eigenvectors. Program output is as follows.

```
**ENTERING PJDREVCOM************************************

Computing the eigenvalue(s) closest to  0.00000E+00
-----------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.18 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:   0.00000E+00
   ILU computed:      0.00[sec]
-----------------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
  -4.1813094904623E+00       1.5E-11       2.8E-22       -2(---) (*)
  -1.8829821916247E+00       1.3E-11       9.1E-23       -1(---)
   1.0315023277911E-01       4.3E-11       1.0E-21        1(---)
   1.8777797389543E+00       2.3E-11       3.2E-22        2(---)
   3.4922682206843E+00       3.7E-11       2.0E-21        3(---)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     85 ;  Estimated gap: 7.96E-01 ;  INFO:  0

CPU time for eigenvalue computation:      0.02[sec]
       CPU time for preconditioning:      0.00[sec]
                   Total CPU time:      0.02[sec]

 (Suggested DROPTOL if restart needed: 2.41E-03)

**LEAVING PJDREVCOM************************************
```

The diagnostic output tells you that initially an ILU using your given `DROPTOL=1.0D-3` is computed. The fill-in factor here refers to how much of `MEM` was really used. In this example, apparently `MEM=20.0` is much more than you really needed. The output also informs you that the ILU has approximated $A - sI$, where the shift parameter s is equal to 0.0 in this example. In addition the computing time is printed. At the end of the eigenvalue computation you will be told about the eigenvalues that have been computed as well as the norm of the associated eigenvector residual $\|A\,\mathbf{u} - \theta\,\mathbf{u}\|$ ("Res. norm"), which can also be found in output vector `RES`. In addition, for each eigenvalue JADAMILU will provide a bound ("Error est.") on the distance to the closest exact eigenvalue. Note that this bound does not take into account rounding error effects, which are of the order $\varepsilon_{\text{mach}}\|A\|$, where $\varepsilon_{\text{mach}}$ is the unit roundoff of the machine, and where $\|A\|$ (the norm of A) is here equal to its largest eigenvalue in modulus, see [4] for details. Hence, in practice, the error bound is the maximum of the indicated error estimate and $\varepsilon_{\text{mach}}\|A\|$ (see Section 6 for additional remarks on the error bound). This information is

followed by the number of the eigenvalue with, in brackets, the index of the cluster to which it belongs, if any. Eigenvalues are sorted in increasing order, and eigenvalues less than the target are assigned a negative number, the closest to the target having number -1, whereas eigenvalues greater than the target are assigned positive numbers, starting from $+1$. When eigenvalues are close to each other, JADAMILU treats them as members of a cluster, which implies that the error bound is computed in a special way. If the distance between eigenvalues in a cluster is below the maximum of the printed error bound and $\varepsilon_{\text{mach}}\|A\|$, it is likely that the eigenvalue is multiple.

This information is followed by the number of matrix-vector multiplications, which corresponds in some sense to the number of iteration steps. Also the gap between the computed eigenvalues and the remaining spectrum is given; `INFO=0` means successful termination. Some timings and suggested drop tolerance finalize the output. We refer to Section 7 for a more detailed description of diagnostic output .

If you want to compute the **smallest eigenvalues**, just use a different value for `ISEARCH`.

Listing 3: EXAMPLE2: Simple driver to compute the smallest eigenvalues

```
c
c     ...
c     changes with respect to Listing 2
c     ...
c     compute the smallest eigenvalues and no a priori information on
c     the eigenvalues is available
      ISEARCH=0
c
c     SIGMA need not to be set in this case
c
c     ...
c
```

Program output is then as follows.

```
**ENTERING PJDREVCOM************************************

Computing the smallest eigenvalue(s)
------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.06 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:  -9.00000E+00
   ILU computed:      0.00[sec]
-----------------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.25 (=nnz(LU)/nnz(A))
   used droptol:   1.25000E-04
     used shift:  -9.00000E+00
   ILU computed:      0.00[sec]
-----------------------------------------------------------
```

```
- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.50 (=nnz(LU)/nnz(A))
   used droptol:   1.56250E-05
     used shift:  -9.00000E+00
   ILU computed:      0.00[sec]
-----------------------------------------------------

           Eigenvalue     Res. norm    Error est.   No(Cluster)
  -7.0552450404553E+00       6.9E-12       1.6E-23        1(---)
  -4.1813094904623E+00       3.9E-11       6.8E-22        2(---)
  -1.8829821916247E+00       2.4E-11       3.0E-22        3(---)
   1.0315023277911E-01       7.7E-11       3.4E-21        4(---)
   1.8777797389543E+00       7.2E-11       3.0E-21        5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:    124 ;  Estimated gap: 1.75E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.02[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.02[sec]

 (Suggested SHIFT   if restart needed:-9.00E+00)
 (Suggested DROPTOL if restart needed: 1.56E-05)

**LEAVING PJDREVCOM************************************
```

Beside the eigenvalues that are computed you will notice that the ILU has been recomputed some steps later with a smaller drop tolerance. This is one of the features of the adaptive preconditioning that is is part of JADAMILU. According to some heuristic rules, JADAMILU will recompute automatically the ILU with different parameters if necessary.

Note that here the shift parameter s such that the ILU approximates $A - sI$ does no more corresponds to `SIGMA`, but is determined by JADAMILU according some heuristic rules, and update if necessary. Note also, that beside a suggested drop tolerance, you also get on output a suggested shift parameter, which may be used in combination with `ISEARCH=1` to speedup somewhat the computation of the smallest eigenvalues, see Section 3.2 below.

3 Detailed Description

As seen in the previous section, JADAMILU offers essentially two different strategies to compute eigenvalues

1. The computation of some eigenvalues at the lower end of the spectrum
2. The computation of a few eigenvalues near a desired target `SIGMA`

If you want to compute eigenvalues at the upper end of the spectrum, you can simply pass $-A$ and compute its eigenvalues at the lower end.

3.1 The easy way to compute the smallest eigenvalues

Without any knowledge about the eigenvalues, simply set `ISEARCH=0`. This is precisely the situation considered in Section 2.

3.2 Advanced computation of the smallest eigenvalues

As you may have seen in Section 2, computing the smallest eigenvalues in principle is easy. But if you have some information available about the location of the eigenvalues then you may supplement JADAMILU with that information.

Consider the matrix from Section 2. We know from the numerical results that the smallest eigenvalue is approximately -7. This is maybe not realistic since we have looked at the computed eigenvalues, but in practice it might be that one has some estimates from the underlying physical problem. Why not providing JADAMILU with this external information? To indicate this we set `ISEARCH=1` and we have to provide an estimate `SIGMA` for the smallest eigenvalue. One then needs to indicate as well a `SHIFT` parameter s, telling that one wants a preconditioner that approximates $A - sI$. The ideal shift is slightly less than the smallest eigenvalue, and a rule of thumb consist in setting the gap between the shift and the smallest eigenvalue roughly equal to the distance between the two smallest eigenvalues. However, is one has no information about this gap, one may use `SHIFT` equal to `SIGMA`.

Listing 4: EXAMPLE3: Advanced driver to compute the smallest eigenvalues

```
c
c     ...
c     changes with respect to Listing 2
c     ...
c     compute the smallest eigenvalues and pass an appropriate SHIFT &
c     SIGMA
      ISEARCH=1
c     Suppose that we know a good estimate for the smallest eigenvalue,
      SIGMA=-7
c     use a shift close to SIGMA but maybe not an exact eigenvalue
      SHIFT=-7
c     ...
c
```

Here is an excerpt of program output.

```
...
- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.06 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:  -7.00000E+00
   ILU computed:      0.00[sec]
ILU-> pos. def.:         1 negative diag. entries changed to positive
...
```

The given shift is actually too large, and the main effect is that the computed ILU preconditioner has some negative eigenvalues, that is, some negative entries on the diagonal of its factors, often referred to as "pivots". Then the preconditioner is not positive definite, whereas it should be such when computing the smallest eigenvalues. As seen from the line (`ILU-> pos. def.: ...`). this is detected and cured by JADAMILU, anyway. JADAMILU converts up to 1% negative pivots into positive ones (default choice `ICNTL(3)=0`). Nevertheless, we recommend to avoid this situation if possible, using a smaller shift (without being too far from the smallest eigenvalue). This is illustrated in EXAMPLE4; running it, you will however discover that this recommended approach, which is based on heuristics, does not necessarily entails faster convergence. You may also turn the conversion by setting `ICNTL(3)=1`. This forces JADAMILU to search for a sufficiently small shift, at the price of extra preprocessing cost.

3.3 Computing eigenvalues close to a given target

Here you have to set `ISEARCH=2` and `SIGMA` to your desired area of interest. This is illustrated in detail in Section 2.

3.4 Dense matrices or matrices that are not explicitly available

Many practical application problems lead to large sparse matrices. Often enough, matrix entries are not given explicitly, and only a matrix-vector multiplication routine is available. Hence the matrix cannot be passed to JADAMILU. A similar situation arises if the matrix is dense or nearly dense, hence storing it in the prescribed format would require excessive memory.

Consider the following example.

$$A = \begin{pmatrix} 0.1 & 5.1 & 0.1 & 0.1 & 0.1 \\ 5.1 & 0.1 & 5.1 & 0.1 & 0.1 \\ 0.1 & 5.1 & 0.1 & 5.1 & 0.1 \\ 0.1 & 0.1 & 5.1 & 0.1 & 5.1 \\ 0.1 & 0.1 & 0.1 & 5.1 & 0.1 \end{pmatrix} = \begin{pmatrix} 0 & 5 & & & \\ 5 & 0 & 5 & & \\ & 5 & 0 & 5 & \\ & & 5 & 0 & 5 \\ & & & 5 & 0 \end{pmatrix} + 0.1 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The matrix is dense, but, certainly we have a fast matrix vector multiplication available, since we have a sparse matrix plus a low rank correction. For the preconditioner one can simply use the sparse part. The ideal JADAMILU driver for this situation is `PJDREVCOM` since it only requires an **approximation** `A,IA,JA` to the original matrix, which will be used to build the preconditioner.

Since for the matrix-vector multiplication one has to use the original dense matrix which cannot be passed to JADAMILU, `PJDREVCOM` returns every time it is needed to perform a matrix-vector multiplication, using a so-called reverse communication protocol. The calling program has then to perform this matrix-vector multiplication with prescribed vectors, and return to `PJDREVCOM`. Note that upon return to the calling program, `PJDREVCOM` indicates in the variable `IJOB` what has to be done, in this case perform matrix-vector multiplication or exit.

Listing 5: EXAMPLE5: Advanced driver using reverse communication

```
      INTEGER    N, MAXEIG, MAXSP
      PARAMETER (N=1000,MAXEIG=5,MAXSP=20)
```

```
      INTEGER    LX
      PARAMETER (LX=N*(3*MAXSP+MAXEIG+1)+4*MAXSP*MAXSP)
      DOUBLE PRECISION   EIGS(MAXEIG), RES(MAXEIG), X(LX)
      INTEGER            NEIG, MADSPACE, ISEARCH, NINIT, ICNTL(5)
      INTEGER            ITER, IPRINT, INFO, IJOB, NDX1, NDX2
      DOUBLE PRECISION   SIGMA, TOL, SHIFT, GAP, MEM, DROPTOL, VAL
c
c     ...
c     declare approximation IA, JA and A and initialize matrix
c     ...
c
c     standard diagnostic report on standard output
      IPRINT=6
c     compute the smallest eigenvalues and no a priori information on
c     the eigenvalues is available
      ISEARCH=0
c     elbow space factor for the fill computed during the ILU
      MEM=20.0
c     tolerance for discarded fill
      DROPTOL=1.d-3
c     number of wanted eigenvalues
      NEIG=MAXEIG
c     no initial approximate eigenvectors
      NINIT=0
c     desired size of the search space
      MADSPACE=MAXSP
c     maximum number of iteration steps
      ITER=1000
c     tolerance for the eigenvector residual
      TOL=1.0d-10
c     additional parameters set to default
      ICNTL(1)=0
      ICNTL(2)=0
      ICNTL(3)=0
      ICNTL(4)=0
      ICNTL(5)=0
c
c     Here we start the computation
      IJOB=0
c
c     We do not have the exact matrix, which A plus a rank one correction;
c     we call DPJDREVCOM (double precision version)
 10   CALL DPJDREVCOM( N,A,JA,IA, EIGS, RES, X, LX, NEIG,
     +                 SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +                 SHIFT, DROPTOL, MEM, ICNTL, IJOB, NDX1, NDX2,
     +                 IPRINT, INFO, GAP)
c
      IF (IJOB.NE.1) GOTO 20
c
c     IJOB=1: MATVEC including rank-1 correction
c         X(NDX1) input, X(NDX2) output
c   (1) MATVEC without rank-1
          call matvec(N,IA,JA,A,X(NDX1),X(NDX2))
c   (2) rank-1 update
c       inner product needed by the rank-1 correction
```

```
          VAL=0.0
          do I=0,N-1
             VAL=VAL+X(NDX1+I)
          end do
c         scaling factor
          VAL=VAL*0.1
c         perform update
          do I=0,N-1
             X(NDX2+I)=X(NDX2+I)+VAL
          end do
        GOTO 10
c
 20     CONTINUE
c
c       release internal memory
        CALL DPJDCLEANUP
```

Reading the program output below you will notice that JADAMILU computes the ILU a few times with smaller and smaller drop tolerance, trying to improve the factorization. So passing an approximate matrix to JADAMILU should be handled with care if the approximation error between the matrix you pass and the matrix used for matrix-vector multiplication is relatively large. There is, however, a way of preventing JADAMILU from recomputing the ILU, which consists in setting `ICNTL(2)=1`, see Section 4.1.

```
**ENTERING PJDREVCOM*************************************

Computing the smallest eigenvalue(s)
-----------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.57 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:  -1.00000E+01
   ILU computed:      0.00[sec]
------------------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.57 (=nnz(LU)/nnz(A))
   used droptol:   1.25000E-04
     used shift:  -1.00000E+01
   ILU computed:      0.01[sec]
------------------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.57 (=nnz(LU)/nnz(A))
   used droptol:   1.56250E-05
     used shift:  -1.00000E+01
   ILU computed:      0.00[sec]
------------------------------------------------------------
```

```
           Eigenvalue     Res. norm    Error est.   No(Cluster)
  -9.9999507505666E+00       4.7E-12       1.5E-19        1(---)
  -9.9998030024238E+00       9.3E-11       5.8E-17        2(---)
  -9.9995567580102E+00       4.6E-11       8.8E-18        3(---)
  -9.9992120174570E+00       5.5E-11       8.9E-18        4(---)
  -9.9987687884203E+00       6.8E-11       1.0E-17        5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     72 ;  Estimated gap: 5.42E-04 ;  INFO:  0

CPU time for eigenvalue computation:      0.01[sec]
       CPU time for preconditioning:      0.01[sec]
                   Total CPU time:      0.02[sec]

 (Suggested SHIFT   if restart needed:-1.00E+01)
 (Suggested DROPTOL if restart needed: 1.56E-05)

**LEAVING PJDREVCOM************************************
```

Note that when slow convergence is detected, JADAMILU tries to exploit the given memory as much as possible to improve the preconditioner. The smallest `DROPTOL` that is used by JADAMILU is $\sqrt{\varepsilon_{\text{mach}}}$; i.e., the square root of the machine precision. Also, no smaller `DROPTOL` can be used once the memory limit is reached, as specified in the variable `MEM`.

3.5 Iteration parameters

On input, you need to set in `ITER` the maximal number of iterations to be performed. If you are sure that you use JADAMILU in the right way, just set `ITER` large enough so that all desired eigenvalues can be computed. Otherwise, in test phase, `ITER` may be use as safeguard.

Another parameter that may have influence on the number of iteration steps is `MADSPACE`. This parameter describes the size of the search space beside the eigenvectors that need to be computed. We recommend to use `MADSPACE` = 20. Choosing a larger number (say 25) might reduce the overall number of iteration steps but at the cost of more expensive single steps which in the end does not necessarily pay off. A smaller `MADSPACE` may be needed if only few memory is available for JADAMILU, as `MADSPACE` influences the minimal size of the array `X` that has to be passed to driver routines. On the other hand, for optimal error control, `MADSPACE` should not be smaller than the maximal number of eigenvalues in a cluster, see Section 6 below.

4 More on preconditioning

As written in the Introduction, JADAMILU uses preconditioning; that is, an approximation M to the shifted matrix $(A - sI)$ (or $A - sB$ in case of a generalized eigenvalue problem). This approximation should be such that systems of type $Mx = y$ are easy to solve, and JADAMILU

uses in fact a multilevel approximate LU factorization of $(A - sI)$ (or $A - sB$). Some more details on its usage are given in this section.

4.1 Adaptive and static preconditioners

By default, JADAMILU recomputes the approximate factorization of $(A - sI)$ as soon as it turns out that outer iterations slow down. As a rule of thumb, first the shift parameter s is adjusted to be closer to the desired eigenvalues. Once the shift is not altered anymore, as second parameter the drop tolerance is reduced. Note however, if you do not want JADAMILU to adaptively recompute the preconditioner, you can turn off adaptivity by setting `ICNTL(2)=1`. Occasionally static preconditioning might be useful, e.g. if only a crude approximation to the exact matrix can be passed to the preconditioner and reducing the drop tolerance will not change that. We consider the matrix EXAMPLE5 from Section 3.4, turning off adaptivity.

Listing 6: EXAMPLE6: Advanced driver using reverse communication

```
c
c      ...
c      changes with respect to Listing 5
c      ...
c      turn off adaptive preconditioning
       ICNTL(2)=1
c      additional parameters set to default
       ICNTL(1)=0
       ICNTL(3)=0
       ICNTL(4)=0
       ICNTL(5)=0
c      ...
```

In the program output below, you will notice that the preconditioner is computed only once and kept throughout the computation. Since the number of iterations remains the same, this turns out to be more efficient in this example.

```
**ENTERING PJDREVCOM**********************************

Computing the smallest eigenvalue(s)
-----------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.57 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:  -1.00000E+01
   ILU computed:      0.00[sec]
------------------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
  -9.9999507505666E+00       4.7E-12       1.5E-19        1(---)
  -9.9998030024238E+00       9.3E-11       5.8E-17        2(---)
  -9.9995567580102E+00       4.6E-11       8.8E-18        3(---)
  -9.9992120174570E+00       5.5E-11       8.9E-18        4(---)
```

```
 -9.9987687884203E+00       6.8E-11       1.0E-17        5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     72 ;  Estimated gap: 5.42E-04 ;  INFO:  0

CPU time for eigenvalue computation:      0.02[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.02[sec]

 (Suggested SHIFT   if restart needed:-1.00E+01)
 (Suggested DROPTOL if restart needed: 1.00E-03)

**LEAVING PJDREVCOM************************************
```

4.2 Recycling the preconditioner and the eigenvectors

In some applications it might happen that you need to compute sequences of eigenvalue problems, maybe even with a changing matrix. Another situation could be that some of your eigenvectors need to be computed with higher accuracy than others. In any case you are faced with using a routine like `PJD` or `PJDREVCOM` a couple of times. If from one call to the next call of, say `PJD`, the matrices are close enough to each other, why not saving time and recycling the old preconditioner as long as possible? This is certainly a situation where JADAMILU is designed for. Furthermore, the user does not need to do anything else but restarting `PJD` with the new matrix and changing `ICNTL(2)=-2`. If the ILU is no longer sufficient JADAMILU will automatically detect it and recompute it based on the matrix you passed to `PJD`.

By way of illustration, consider the matrix

$$A(t) = \begin{pmatrix} 0 & 5 & & & \\ 5 & t & 5 & & \\ & 5 & 2t & 5 & \\ & & 5 & 3t & \ddots \\ & & & \ddots & \ddots \end{pmatrix},$$

for $t = 1.00\,,\ 0.05\,,\ 1.10\,,\ \ldots$ and suppose we are seeking for the eigenvalues near the origin.

Listing 7: EXAMPLE7: Repeated use of JADAMILU

```
c
c     ...
c     changes with respect to Listing 2
c     ...
c
      do L=0,2
c        dynamically create matrix entries
         T=1.0D0+5.0d-2*L
         write (6,'(A,1P,E12.4)') 't=',T
         K=1
         DO I=1,N
```

```
            IA(I)=K
            JA(K)=I
            A(K)=T*DBLE(I-1)
            K=K+1
            IF (I .LT. N) THEN
               JA(K)=I+1
               A(K)=5.0D0
               K=K+1
            END IF
         END DO
         IA(N+1)=K
c
c        always re-init the following parameters since they are
c        overwritten by DPJD
         ITER=1000
c        (ITER set by DPJD to number of iteration steps actually needed)
         NEIG=MAXEIG
c        (NEIG set by DPJD to number of eigenvalues effectively computed)
         CALL DPJD( N,A,JA,IA, EIGS, RES, X, LX, NEIG,
     +              SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +              SHIFT, DROPTOL, MEM, ICNTL,
     +              IPRINT, INFO, GAP)

c        next time we enter DPJD, we continue with adaptive
c        preconditioning based on the preconditioner that has been used
c        so far
         ICNTL(2)=-2
c        use previously computed eigenvectors as initial approximation
         NINIT=NEIG
      end do
c
c     release internal memory and discard preconditioner
      CALL DPJDCLEANUP
```

As you can see in the program output, the preconditioner is not recomputed in this example. Note that this can be enforced by setting `ICNTL(2)=-1`. Then, one re-enters `PJD` with static preconditioning.

```
**ENTERING PJDREVCOM************************************

Computing the eigenvalue(s) closest to  0.00000E+00
-----------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.18 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:   0.00000E+00
   ILU computed:      0.00[sec]
-----------------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
```

```
  -4.1813094904623E+00       1.5E-11       2.8E-22       -2(---) (*)
  -1.8829821916247E+00       1.3E-11       9.1E-23       -1(---)
   1.0315023277911E-01       4.3E-11       1.0E-21        1(---)
   1.8777797389543E+00       2.3E-11       3.2E-22        2(---)
   3.4922682206843E+00       3.7E-11       2.0E-21        3(---)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     85 ;  Estimated gap: 7.96E-01 ;  INFO:  0

CPU time for eigenvalue computation:      0.02[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.02[sec]

 (Suggested DROPTOL if restart needed: 2.41E-03)

**LEAVING PJDREVCOM************************************

t=  1.0500E+00

**ENTERING PJDREVCOM************************************

Computing the eigenvalue(s) closest to  0.00000E+00
--------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
  -4.0120453498900E+00       6.0E-11       2.6E-21       -2(---) (*)
  -1.6436284668853E+00       7.7E-11       2.9E-21       -1(---)
   4.0080605221695E-01       9.6E-11       5.0E-21        1(---)
   2.2251865872115E+00       7.9E-11       3.7E-21        2(---)
   3.8823846177623E+00       5.5E-11       2.3E-20        3(---)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     53 ;  Estimated gap: 1.39E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.00[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.00[sec]

 (Suggested DROPTOL if restart needed: 1.05E-02)

**LEAVING PJDREVCOM************************************

t=  1.1000E+00
```

```
**ENTERING PJDREVCOM************************************

Computing the eigenvalue(s) closest to  0.00000E+00
-------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
  -3.8466403861287E+00       6.5E-11       1.0E-20       -2(---)
  -1.4096605995227E+00       2.6E-11       3.1E-22       -1(---)
   6.9159556864751E-01       1.1E-10       6.1E-21        1(---)
   2.5642166391883E+00       3.7E-11       8.2E-22        2(---)
   4.2625293553861E+00       4.3E-11       1.2E-21        3(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     57 ;  Estimated gap: 1.56E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.01[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.01[sec]

 (Suggested DROPTOL if restart needed: 4.86E-02)

**LEAVING PJDREVCOM************************************
```

In listing 7, you can see that some parameters are redefined between successive calls to `PJD`, because they are overwritten on output. Here is a list of input parameters overwritten on output.

- `ITER`. On input: maximum number of iteration steps; on output: number of steps needed by `PJD`. *Need always to be reinitialized*
- `DROPTOL`. On input: initial drop tolerance for the ILU; on output: suggested drop tolerance for next run. Hence one may proceed the computation with this new value.
- `NEIG`. On input: number of desired eigenvalues; on output: number of computed eigenvalues. *Need to be reinitialized since it may happen that not all eigenvalues wanted have been computed successfully.*
- `EIGS,X`. On input: if `NINIT`> 0, approximate eigenvalues and eigenvectors (see Section 5 for details); on output: computed eigenvalues and eigenvectors. Hence one may proceed the computation with these, and further set `NINIT=NEIG` to use them as initial guess for the next run.
- `SIGMA,SHIFT` when `ISEARCH`≤ 1. On input, if `ISEARCH`$= 1$: values supplied by the user for initial eigenvalue approximation and initial shift; on output: parameters suggested by `PJD` for next run. Hence one may proceed the computation with this new values, and further reset `ISEARCH`$= 1$ if one had initially `ISEARCH`$= 0$.

Remark. Although it sounds reasonable in a parameter-dependent problem to use the old eigenvectors as initial guesses, this should be considered with care. Indeed, JADAMILU uses

a Newton like process; i.e., it favors convergence towards the solutions in the vicinity of the provided initial guesses. But, as the computation proceeds, these solutions may be no more those really wanted; that is, the smallest or those closest to the given target. For instance, if you run an example similar to EXAMPLE7 but with a matrix $A = t\,I + C$, where C is some symmetric matrix, you will be happy to see that once JADAMILU has computed eigenvalues & eigenvectors for some t, reentering JADAMILU with a different t gives you the answer with minimal number of matrix vector multiplications. In fact, any inner iteration is skipped, JADAMILU just computes the eigenvalues associated to these initial eigenvectors and the corresponding residual norms. Since they are below `TOL`, JADAMILU has found solutions to the eigenvalue problem that are accurate enough and exits. But then JADAMILU has no mean to check that the computed eigenvalues are the wanted one. This can be only ensured by restarting the computation from scratch.

4.3 Some advanced preconditioning parameters

Here we will comment on some parameters that you could vary from the default settings if it would improve your application.

`DROPTOL`

We suggest to start with a value `DROPTOL=1.0d-3`, but certainly you may use any value less than 1 based on your experience. `DROPTOL` is roughly spoken the drop tolerance which defines "small" entries in absolute values. JADAMILU essentially computes an approximate factorization $LDL^T \approx A$ such that L is unit lower triangular (some details concerning multilevel strategies are skipped here for simplicity). During the approximate factorization, in any column L of the approximate factorization, entries less than `DROPTOL` in absolute value are dropped (beside some additional safeguard strategies). Note that the drop tolerance is interpreted in a relative sense, hence it needs not be adapted according to the scaling of the matrix.

Note also that JADAMILU is designed to use adaptive preconditioning, i.e., if the drop tolerance is too rough and if enough memory is available, then the drop tolerance is reduced and a new ILU is computed. So starting with a drop tolerance that is too rough will delay the computation for some time. On the other hand if you start with a drop tolerance that is (maybe) chosen too small, JADAMILU keeps it as long as the preconditioner fits into the memory. So choosing a relatively small drop tolerance should be considered with care. If the drop tolerance is too small to factor the approximate LU within the given memory, JADAMILU automatically increases `DROPTOL` in case of adaptive preconditioning.

`MEM`

`MEM` describes the memory relative to the number of nonzero entries of the original matrix that the user is willing to spend. E.g. `MEM`= 20 means that the ILU is allowed to consume 20 times the memory of the original matrix. In general, using a sufficiently large `MEM` parameter is recommended. In particular because this allows JADAMILU to use adaptive preconditioning and reduce the drop tolerance further if necessary. Please note that since JADAMILU computes a multilevel ILU, it may happen that the final fill-in that is printed on the screen is less than the memory that was required in intermediate steps. In particular you should not be confused

that adaptive preconditioning does not compute a new preconditioner, although you may think that there should be enough space left.

`ICNTL(2)`

We have already seen that playing around with `ICNTL(2)` allows adaptive preconditioning (`ICNTL(2)`$= 0$, default), static preconditioning (`ICNTL(2)`$= 1$), re-enter with adaptive preconditioning (`ICNTL(2)`$= -2$) and re-enter with static preconditioning (`ICNTL(2)`$= -1$).

`ICNTL(3)`

Another advanced option that you may alter is given by `ICNTL(3)`. In case of seeking for the smallest eigenvalues (`ISEARCH=0,ISEARCH=1`), the shifted matrix $A - sI$ is expected to be positive definite, if the `SHIFT` parameter is sufficiently small. However, it may happen that s is not accurate enough or simply too big. In that case, $A - sI$ will no longer stay positive definite and it is likely that the ILU will encounter negative or indefinite pivots. By default, JADAMILU allows up to 1% negative diagonal diagonal pivots that will be converted to their positive definite counterparts. By setting `ICNTL(3)=1` this option is disabled and you force JADAMILU to compute a sufficiently small shift.

`ICNTL(4)`

A rather delicate parameter is `ICNTL(4)`. This parameter is directly connected to the multilevel ILU. In principle the computed ILU includes a strategy to compute $A \approx LDL^T$, where L is unit lower triangular and D is block diagonal, such that $\|L^{-1}\|_\infty \leq 5$. The bound 5 is the default value and since it may happen that this bound is too small, JADAMILU has an internal procedure to raise this bound while it is computing the preconditioner. You may in any case overwrite the initial bound 5 by assigning your desired bound to `ICNTL(4)`, i.e. `ICNTL(4)=100` will raise the initial bound to 100 instead of 5.

`ICNTL(5)`

The parameter `ICNTL(5)` has recently been added to JADAMILU starting from release 2.0. This parameter is meaningless if single precision is used. For double precision, this parameter is used to turn on mixed precision computing. Whenever ICNTL(5) is equal to 1, the computation and the application of the preconditioner is done in single precision only. This should not affect the final accuracy since everything else is kept using double precision.

Note that in this case the `MEM` parameter refers to the memory available for the ILU relative to the memory of the original matrix as if it were stored in single precision; that is, for the same physical memory available, you may use `MEM` twice as large when `ICNTL(5)`$= 1$.

4.4 Custom preconditioners

If you prefer not to use JADAMILU's built-in preconditioner, you can use the routine `JDREVCOM` to apply a private preconditioning routine. Suppose your matrix is given by

$$A = \begin{pmatrix} 4 & -1 & & & & & -1 \\ -1 & 4 & -1 & & & & \\ & -1 & 4 & -1 & & & \\ & & -1 & 4 & -1 & & \\ & & & -1 & \ddots & \ddots & \\ & & & & \ddots & \ddots & -1 \\ -1 & & & & & -1 & 4 \end{pmatrix}$$

and you would prefer only to use

$$A^{-1} \approx B = \frac{1}{64} \begin{pmatrix} 16 & 4 & 1 & & & 1 & 4 \\ 4 & 16 & 4 & 1 & & & 1 \\ 1 & 4 & 16 & 4 & 1 & & \\ & 1 & 4 & 16 & 4 & 1 & \\ & & 1 & 4 & \ddots & \ddots & \ddots \\ 1 & & & 1 & \ddots & 16 & 4 \\ 4 & 1 & & & \ddots & 4 & 16 \end{pmatrix}$$

as direct sparse approximate inverse preconditioner. In that case you simply have to provide your own matrix-vector multiplication subroutine together with your private preconditioner. The driver `JDREVCOM` uses reverse-communication and it stops every time when you have to apply your custom routines. After that you restart `JDREVCOM` and it will resume with your data.

Listing 8: EXAMPLE8: Advanced driver using reverse communication

```
      INTEGER     N, MAXEIG, MAXSP
      PARAMETER (N=1000,MAXEIG=5,MAXSP=20)
      INTEGER     LX
      PARAMETER (LX=N*(3*MAXSP+MAXEIG+1)+4*MAXS P*MAXSP)
      DOUBLE PRECISION   EIGS(MAXEIG), RES(MAXEIG), X(LX)
      INTEGER            NEIG, MADSPACE, ISEARCH, NINIT, ICNTL(5)
      INTEGER            ITER, IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, SHIFT, GAP, MEM, DROPTOL
      INTEGER            IJOB, NDX1, NDX2
c
c     ...
c
c     standard diagnostic report on standard output
      IPRINT=6
c     we want the smallest eigenvalues
      ISEARCH=0
c     number of wanted eigenvalues
      NEIG=MAXEIG
c     no initial approximate eigenvectors
      NINIT=0
```

```
c      desired size of the search space
       MADSPACE=MAXSP
c      maximum number of iteration steps
       ITER=1000
c      tolerance for the eigenvector residual
       TOL=1.0d-10
c      additional parameters set to default
       ICNTL(1)=0
       ICNTL(2)=0
       ICNTL(3)=0
       ICNTL(4)=0
       ICNTL(5)=0
c
c      start reverse communication
       IJOB=0
 10    CALL JDREVCOM( N, EIGS, RES, X, LX, NEIG,
     +               SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +               IJOB, NDX1, NDX2, IPRINT, INFO, GAP)
c
c      your private matrix-vector multiplication
       IF (IJOB.EQ.1) THEN
c         X(NDX1) input,  X(NDX2) output
          X(NDX2)=4*X(NDX1)-X(NDX1+1)-X(NDX1+N-1)
          DO I=1,N-2
             X(NDX2+I)=-X(NDX1+I-1)+4*X(NDX1+I)-X(NDX1+I+1)
          END DO
          X(NDX2+N-1)=-X(NDX1)-X(NDX1+N-2)+4*X(NDX1+N-1)
          GOTO 10
c      your private preconditioner
       ELSE IF (IJOB.EQ.2) THEN
c         X(NDX2) input,  X(NDX1) output
          X(NDX1)  =(16*X(NDX2)+ 4*X(NDX2+1)+  X(NDX2+2)+X(NDX2+N-2)
     +              +4*X(NDX2+N-1))/64.0
          X(NDX1+1)=( 4*X(NDX2)+16*X(NDX2+1)+4*X(NDX2+2)+X(NDX2+3)
     +              +X(NDX2+N-1))/64.0
          DO I=2,N-3
             X(NDX1+I)=(X(NDX2+I-2)+4*X(NDX2+I-1)+16*X(NDX2+I)
     +              +4*X(NDX2+I+1)+X(NDX2+I+2))/64.0
          END DO
          X(NDX1+N-2)=(X(NDX2)+X(NDX2+N-4)+4*X(NDX2+N-3)
     +            +16*X(NDX2+N-2)+4*X(NDX2+N-1))/64.0
          X(NDX1+N-1)=(4*X(NDX2)+X(NDX2+1)+X(NDX2+N-3)
     +              +4*X(NDX2+N-2)+16*X(NDX2+N-1))/64.0
          GOTO 10
       END IF
c
c      work done - release internal memory
       CALL DPJDCLEANUP
```

When executing this program, you will notice that two out of five eigenvalues could not be approximated to full precision. Certainly, since there is no adaptive preconditioning, there is no way to improve your preconditioner and thus you should carefully think about the options (e.g. is ITER= 1000 still appropriate?).

4.5 Diagonal preconditioning

In some applications, the matrix may not be explicitly available besides its diagonal entries. When dealing with extremely large systems, it may also happen that no memory is left for computing an ILU preconditioning. To face these situations, JADAMILU offers the possibility to use only diagonal preconditioning. To do this you simply pass `IA,JA,A` to `PJDREVCOM`, but by setting `JA(1)` to a value less than 0, e.g. `JA(1)=-1`, you indicate that your matrix is diagonal. In this case, the other entries of `JA` as well as `IA` are not referenced. Only `A(1),...,A(N)` are needed. Here it is important to note that adaptive preconditioning will access the diagonal matrix from time to time in order to set up a new diagonal preconditioner based on a potential new shift parameter. The `MEM` and `DROPTOL` parameter are not referenced. Consider as example the matrix

$$A = \begin{pmatrix} 1 & 5 & & & \\ 5 & 2 & 5 & & \\ & 5 & 3 & 5 & \\ & & 5 & 4 & \ddots \\ & & & \ddots & \ddots \end{pmatrix}$$

and suppose we are seeking for the smallest eigenvalues.

Listing 9: EXAMPLE9: Compute the smallest eigenvalues using diagonal preconditioning

```
      INTEGER    N, MAXEIG, MAXSP
      PARAMETER (N=1000,MAXEIG=5,MAXSP=20)
*     optimal workspace (here MAXSP*MAXSP>MAXEIG)
      INTEGER LX
      PARAMETER (LX=N*(3*MAXSP+MAXEIG+1)+4*MAXSP*MAXSP)
      DOUBLE PRECISION   EIGS(MAXEIG), RES(MAXEIG), X(LX), D(N)
*     arguments to pass to the routines
      INTEGER            NEIG, MADSPACE, ISEARCH, NINIT, ICNTL(5)
      INTEGER            ITER, IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, GAP,MEM, DROPTOL,SHIFT
      INTEGER            IJOB, NDX1, NDX2, I,J,K
c
c     ...
c     declare IA, JA and A and initialize matrix
c     ...
c
c     use a copy of the diagonal entries to be passed to DJDREVCOM
      DO I=1,N
         D(I)=0.0
         DO J=IA(I),IA(I+1)-1
            K=JA(J)
            IF (K.EQ.I) D(I)=A(J)
         END DO
      END DO
c
c     standard diagnostic report on standard output
      IPRINT=6
c     we want the smallest eigenvalues
      ISEARCH=0
c     number of wanted eigenvalues
      NEIG=MAXEIG
```

```
c      no initial approximate eigenvectors
       NINIT=0
c      desired size of the search space
       MADSPACE=MAXSP
c      maximum number of iteration steps
       ITER=1000
c      tolerance for the eigenvector residual
       TOL=1.0d-10
c      additional parameters set to default
       ICNTL(1)=0
       ICNTL(2)=0
       ICNTL(3)=0
       ICNTL(4)=0
       ICNTL(5)=0
c
       IJOB=0
 10    CALL DPJDREVCOM( N,D,-1,-1, EIGS, RES, X, LX, NEIG,
     +                  SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +                  SHIFT, DROPTOL, MEM, ICNTL,
     +                  IJOB, NDX1, NDX2, IPRINT, INFO, GAP)
c
c      your private matrix-vector multiplication
       IF (IJOB.NE.1) GOTO 20
c                     input,  output
       CALL MYMATVEC(X(NDX1),X(NDX2))
       GOTO 10
c
 20    CONTINUE
c
c      release internal memory
       CALL DPJDCLEANUP
```

As you will see from the printed output, PJDREVCOM will continue to adapt the shift and the diagonal preconditioner.

```
**ENTERING PJDREVCOM************************************

Computing the smallest eigenvalue(s)
------------------------------------

- Setting up a new diagonal preconditioner -------------
used shift:  1.0000E-03
-----------------------------------------------------------

- Setting up a new diagonal preconditioner -------------
used shift: -5.9139E+00
-----------------------------------------------------------

- Setting up a new diagonal preconditioner -------------
used shift: -9.9767E+00
-----------------------------------------------------------
```

```
- Setting up a new diagonal preconditioner -------------
used shift: -8.8221E+00
-----------------------------------------------------

- Setting up a new diagonal preconditioner -------------
used shift: -6.2671E+00
-----------------------------------------------------

           Eigenvalue     Res. norm    Error est.   No(Cluster)
 -6.0552450404553E+00       8.7E-11       2.6E-21        1(---)
 -3.1813094904623E+00       7.0E-11       2.1E-21        2(---)
 -8.8298219162471E-01       5.7E-11       1.7E-21        3(---)
  1.1031502327791E+00       4.6E-11       1.2E-21        4(---)
  2.8777797389543E+00       8.7E-11       4.2E-21        5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:    166 ;  Estimated gap: 1.78E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.02[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.02[sec]

 (Suggested SHIFT   if restart needed:-6.06E+00)

**LEAVING PJDREVCOM************************************
```

Certainly you can reuse the diagonal part after you passed the diagonal entries to `PJDREVCOM`. To do this, simply set `ICNTL(1)=2`. By giving away the diagonal part, `PJDREVCOM` will automatically turn off adaptive preconditioning.

Listing 10: EXAMPLE10: Compute the smallest eigenvalues using diagonal preconditioning, saving memory

```
c
c      ...
c      changes with respect to Listing 9
c      ...
c
c      use a copy of the diagonal entries to be passed to PJDREVCOM
       DO I=1,N
          X(I)=0.0
          DO J=IA(I),IA(I+1)-1
             K=JA(J)
             IF (K.EQ.I) X(I)=A(J)
          END DO
       END DO
c
c      ...
c
c      ICNTL(1) set to 2 because the matrix that is passed on input
```

```
c               will be lost and overwritten by the eigenvector
c               approximation X
      ICNTL(1)=2
c     additional parameters set to default
      ICNTL(2)=0
      ICNTL(3)=0
      ICNTL(4)=0
      ICNTL(5)=0
c
      IJOB=0
 10   CALL DPJDREVCOM( N,X,-1,-1, EIGS, RES, X, LX, NEIG,
     +                 SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +                 SHIFT, DROPTOL, MEM, ICNTL,
     +                 IJOB, NDX1, NDX2, IPRINT, INFO, GAP)
c
c     your private matrix-vector multiplication
      IF (IJOB.NE.1) GOTO 20
c                     input,  output
      CALL MYMATVEC(X(NDX1),X(NDX2))
      GOTO 10
c
 20   CONTINUE
c
c     release internal memory
      CALL DPJDCLEANUP
```

As you will notice, in contrast to Listing 9, no adaptive preconditioning is performed and the eigenvalues are computed based on the initial diagonal matrix.

```
**ENTERING PJDREVCOM************************************

Computing the smallest eigenvalue(s)
------------------------------------

- Setting up a new diagonal preconditioner -------------
used shift:  1.0000E-03
-----------------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
-6.055245040455286E+00       7.1E-11       1.8E-21        1(---)
-3.181309490462310E+00       3.3E-11       4.7E-22        2(---)
-8.829821916247098E-01       4.4E-11       9.8E-22        3(---)
 1.103150232779112E+00       8.7E-11       4.3E-21        4(---)
 2.877779738954344E+00       4.0E-11       8.9E-22        5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:    185 ;  Estimated gap: 1.85E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.04[sec]
```

```
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.04[sec]

 (Suggested SHIFT   if restart needed:-6.06E+00)

**LEAVING PJDREVCOM************************************
```

5 Refined eigenvalue computations

A frequent problem in eigenvalue computations occurs, when given eigenvectors have been computed with some low accuracy and you would like to refine the accuracy. Another related problem could be that you may know some crude approximation to some of the eigenvectors based on the physical background.

The simplest example to illustrate this is to run `PJD` with some relatively large `TOL` first and then use `PJD` to improve the accuracy. Moreover it sounds reasonable that when improving the accuracy, you may keep the old preconditioner and do not need adaptive preconditioning anymore. The JADAMILU routines allow you to pass initial approximate eigenvectors. To do this set `NINIT` to the number of eigenvectors you provide and overwrite the first `N*NINIT` components of `X` with your given approximate eigenvectors. Furthermore, as described in Section 4.1 you can re-enter the routine with the old preconditioner and decide whether to continue with adaptive preconditioning (`ICNTL(2)=-2`) or simply to stay static (`ICNTL(2)=-1`). As an example consider A from Section 4.5.

Listing 11: EXAMPLE11: Eigenvector refinement

```
      INTEGER    N, MAXEIG, MAXSP
      PARAMETER (N=1000,MAXEIG=5,MAXSP=20)
*     optimal workspace (here MAXSP*MAXSP>MAXEIG)
      INTEGER LX
      PARAMETER (LX=N*(3*MAXSP+MAXEIG+1)+4*MAXSP*MAXSP)
      DOUBLE PRECISION   EIGS(MAXEIG), RES(MAXEIG), X(LX)
*     arguments to pass to the routines
      INTEGER            NEIG, MADSPACE, ISEARCH, NINIT, ICNTL(5)
      INTEGER            ITER, IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, GAP,MEM, DROPTOL,SHIFT
c
c     ...
c     declare IA, JA and A and initialize matrix
c     ...
c
c     standard report on standard output
      IPRINT=6
c
c     we want the smallest eigenvalues
      ISEARCH=0
c     elbow space factor for the fill computed during the ILU
      MEM=20.0
c     tolerance for discarded fill
      DROPTOL=1.d-3
c     number of wanted eigenvalues
      NEIG=MAXEIG
```

```
c      no initial approximate eigenvectors
       NINIT=0
c      desired size of the search space
       MADSPACE=MAXSP
c      maximum number of iteration steps
       ITER=1000
c      LOW tolerance for the eigenvector residual
       TOL=1.0d-3
c      additional parameters set to default
       ICNTL(1)=0
       ICNTL(2)=0
       ICNTL(3)=0
       ICNTL(4)=0
       ICNTL(5)=0
c
       CALL DPJD(N,A,JA,IA, EIGS, RES, X, LX, NEIG,
     +           SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +           SHIFT, DROPTOL, MEM, ICNTL,
     +           IPRINT, INFO, GAP)
c
c      turn off adaptivity and recycle preconditioner
       ICNTL(2)=-1
c      on input pass the computed eigenvectors
       NINIT=NEIG
c      improve accuracy of the eigenvectors
       TOL=1.0d-10
c      re-init number of iteration steps, because it was overwritten
       ITER=1000
c      remember that SIGMA, SHIFT were returned from PJD, re-use values
       ISEARCH=1
c      number of wanted eigenvalues
       NEIG=MAXEIG
c
       CALL DPJD(N,A,JA,IA, EIGS, RES, X, LX, NEIG,
     +           SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +           SHIFT, DROPTOL, MEM, ICNTL,
     +           IPRINT, INFO, GAP)
c
c      release internal memory and discard preconditioner
       CALL DPJDCLEANUP
```

On execution you will see that `PJD` is called twice, once computing the preconditioner and returning the desired eigenvector according to a low accuracy. The second time `PJD` is invoked, the preconditioner and the old eigenvectors are already present. The accuracy is improved using a smaller `TOL`.

```
**ENTERING PJDREVCOM**********************************

Computing the smallest eigenvalue(s)
-----------------------------------

- Setting up a new ILU preconditioner ------------------
```

```
 fill-in factor:      1.06 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:  -8.00000E+00
   ILU computed:      0.00[sec]
-------------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.25 (=nnz(LU)/nnz(A))
   used droptol:   1.25000E-04
     used shift:  -8.00000E+00
   ILU computed:      0.00[sec]
-------------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.50 (=nnz(LU)/nnz(A))
   used droptol:   1.56250E-05
     used shift:  -8.00000E+00
   ILU computed:      0.00[sec]
-------------------------------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
  -6.0552450402030E+00       1.4E-04       7.0E-09        1(---)
  -3.1813094896192E+00       3.0E-04       3.8E-08        2(---)
  -8.8298218873885E-01       4.8E-04       1.2E-07        3(---)
   1.1031502338478E+00       4.5E-04       1.1E-07        4(---)
   2.8777797469109E+00       5.5E-04       1.8E-07        5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     60 ;  Estimated gap: 1.65E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.01[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.01[sec]

 (Suggested SHIFT   if restart needed:-8.00E+00)
 (Suggested DROPTOL if restart needed: 1.56E-05)

**LEAVING PJDREVCOM************************************


**ENTERING PJDREVCOM************************************

Computing the smallest eigenvalue(s)
-----------------------------------

            Eigenvalue     Res. norm    Error est.   No(Cluster)
```

```
  -6.0552450404553E+00       1.1E-11       3.9E-23       1(---)
  -3.1813094904623E+00       7.7E-11       2.6E-21       2(---)
  -8.8298219162471E-01       1.9E-11       1.8E-22       3(---)
   1.1031502327791E+00       3.6E-11       7.4E-22       4(---)
   2.8777797389543E+00       3.4E-11       6.6E-22       5(---) (*)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)

  Number of MATVEC needed:     68 ;  Estimated gap: 1.86E+00 ;  INFO:  0

CPU time for eigenvalue computation:      0.01[sec]
       CPU time for preconditioning:      0.00[sec]
                    Total CPU time:      0.01[sec]

 (Suggested SHIFT   if restart needed:-8.00E+00)
 (Suggested DROPTOL if restart needed: 8.52E-05)

**LEAVING PJDREVCOM************************************
```

You can verify that calling the same routine only once with `TOL=1.0d-10` from the very beginning will roughly need the same number of matrix-vector multiplications.

6 Accuracy and missconvergence

Whenever you call one of JADAMILU's driver you have to prescribe a parameter `TOL` that describes the desired accuracy. If possible, on exit from JADAMILU the eigenvectors are computed such that the eigenvalue residual $\|Ax-\theta x\|$ (with $\|x\| = 1$) is less than `TOL`. However, accuracy is limited by rounding errors, and if `TOL` is below the attainable accuracy, JADAMILU will detect it and truncate the computation. Roughly, the attainable accuracy is of the order $\varepsilon_{\text{mach}}\|A\|$, where $\varepsilon_{\text{mach}}$ is the unit roundoff of the machine, and where $\|A\|$ (the norm of A) is here equal to its largest eigenvalue in modulus. Using very small `TOL` should however be considered with care. The detection of a too small `TOL` requires several iterations, implying a waste of computing time. It may also fail if `TOL` is by far too small, in which cases JADAMILU stagnates until the maximum number of iterations is reached. The JADAMILU drivers will return on exit `RES` to hold the actual norm of the residuals.

`TOL` is roughly the accuracy of computed eigenvectors. If only the accuracy of eigenvalues matters for you, you are more interested in the printed error bound. You may have noticed that the latter is of the order of the square of `TOL`, i.e., accuracy of eigenvalues is quadratic with respect to the accuracy of eigenvectors. Hence a moderate `TOL` is enough to get full accuracy on the eigenvalues. Eigenvalues for which the error bound is not quadratic are eigenvalues that belong to a cluster that cannot be treated as such by JADAMILU because either its size is too large or because it contains eigenvalues that, among the computed ones, are either the largest ones (cases `ISEARCH=0,1`), or the farthest from the target (case `ISEARCH=2`). In these cases, a better error bound for these eigenvalues will be obtained by increasing the number of computed eigenvalues. On the other hand, the maximal size of a cluster that JADAMILU can treat as such is equal to the `MADSPACE` parameter discussed in Section 3.5.

As already written in Section 2, "Error est." is a a bound on the distance to the closest exact eigenvalue, which, however does not take into account some rounding error effects. Hence in practice, the actual error bound is the maximum of the printed error estimate and $\varepsilon_{\text{mach}}\|A\|$. Note also that the printed bound requires in principle to know where uncomputed eigenvalues are located. Hence it becomes untrustable if missconvergence occurred. *Missconvergence* is a situation where some eigenvalues are missed in the interval between the smallest and the largest of the computed eigenvalues. This may happen incidentally, and missed eigenvalues are in general close to the boundary of the interval. Then, the error bound is wrong for the eigenvalues close to missed eigenvalues, that is, close to the boundary of the interval. Here again, a safeguard consists in computing more eigenvalues than actually needed.

Finally, for either the largest eigenvalue (cases `ISEARCH=0,1`), or the farthest from the target (case `ISEARCH=2`), the error bound depends on the estimated gap between computed and uncomputed eigenvalues. Because this gap is only estimated (and not computed accurately) by JADAMILU, the "Error est." printed for this eigenvalue is no more an error bound, but an indicative estimate.

7 Diagnostic Output

The default value of `IPRINT` is recommended to be set to 6. This refers to standard output. If, instead, IPRINT is set to a positive number (different from 5), the information is printed in the file with unit number equal to the indicated value.

The information that is printed tells you how often the ILU has been (re-)computed, how many eigenvalues have been computed and how accurate they are. To be precise, take Listing 2 (EXAMPLE1) and its output.

```
**ENTERING PJDREVCOM************************************

Computing the eigenvalue(s) closest to  0.00000E+00
-----------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:      1.18 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:   0.00000E+00
   ILU computed:      0.00[sec]
-----------------------------------------------------------

           Eigenvalue     Res. norm    Error est.   No(Cluster)
  -4.1813094904623E+00       1.5E-11       2.8E-22       -2(---) (*)
  -1.8829821916247E+00       1.3E-11       9.1E-23       -1(---)
   1.0315023277911E-01       4.3E-11       1.0E-21        1(---)
   1.8777797389543E+00       2.3E-11       3.2E-22        2(---)
   3.4922682206843E+00       3.7E-11       2.0E-21        3(---)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is indicative only)
```

```
  Number of MATVEC needed:     85 ;  Estimated gap: 7.96E-01 ;  INFO:  0

CPU time for eigenvalue computation:      0.02[sec]
       CPU time for preconditioning:      0.00[sec]
                     Total CPU time:      0.02[sec]

 (Suggested DROPTOL if restart needed: 2.41E-03)

**LEAVING PJDREVCOM************************************
```

You start with `DROPTOL=1e-3`. This leads to an approximate factorization (ILU) of A that has 1.18 as many nonzeros as the initial matrix. Furthermore, the initial shift is chosen to be 0.0; i.e., an ILU of $A - 0.0\,I$ is computed. The computing time is negligible. Since there is no further message concerning any ILU, this will be the only time that an approximate factorization is computed. At the end of the computation you will be faced with the computed eigenvalues. "Res. norm" refers to the norm of the associated eigenvector residual $\|Ax - \theta x\|_2$ in the Euclidean norm, "Error est." is the error bound (see Section 6) and finally "No(Cluster)" tells you the number of the eigenvalue (see Section 2 for the numbering) and the index of the cluster to which the eigenvalue belongs, if any (see Section 6 for more about clustered eigenvalues).

Furthermore, the number of MATVEC (matrix-vector multiplications) can be considered as a measure for the total number of iteration steps. The estimated gap shows you how far your eigenvalues are with respect to the remaining spectrum. `INFO`= 0 indicates successful computation. At the end a computation time statistics is given that splits the total time into the time spent in computing the preconditioner (maybe more than one time) and the remaining eigenvalue computation.

If you are sick of all this printed statements, simply set `IPRINT=0` and any output is suppressed.

If you would like to have very detailed information you can choose `IPRINT=-6` (or a negative number different from -5). In this case you will get detailed information about the multilevel ILU (how many levels, how much fill on each level, ...). In addition, at each outer iteration step you will be told about the progress of the eigenvalue computation.

```
Step: 0(1); #MV:  1; Rayl. quot.: 5.09490E+02; Res. norm: 2.9E+02
Step: 1(1); #MV:  3; Rayl. quot.: 9.13765E-01; Res. norm: 8.3E+00

...

Step: 5(1); #MV: 11; Rayl. quot.: 1.03150E-01; Res. norm: 5.7E-07
Step: 6(1); #MV: 15; Rayl. quot.: 1.03150E-01* converged (  15 MV)
```

"Step" refers to one outer iteration step and in brackets you are told which eigenvector is computed (here: first eigenvector). Next you will be informed about the total number of matrix-vector multiplications so far (#MV). The difference between two neighboring #MV will tell you how many inner steps were required. "Rayl. quot" describes the size of the current approximate eigenvalue based on the Rayleigh quotient $\theta = \frac{x^T A x}{x^T x}$. As soon as the method converges you will see that the Rayleigh quotient does not change anymore. Finally "Res.

norm" is the norm of the associated eigenvector residual. Once JADAMILU has successfully computed an eigenvector, it will notify you the success by summarizing the number matrix-vector multiplication instead of the residual norm.

8 Generalized eigenvalue problems

The usage of drivers for generalized eigenvalue problems is pretty similar to that of the corresponding drivers for standard eigenvalue problems; `PJD_GEP` and `PJDREVCOM_GEP` have, as additional arguments, the arrays `B`, `JB` and `IB` in which one should store the mass matrix B using the same format as for A, described in Appendix B. **Note that B should be positive definite.** Other parameters remain the same and have same meaning.

Similarly to `A`, `JA` and `IA`, when using `PJDREVCOM_GEP`, `B`, `JB` and `IB` need not contain the exact matrix, since matrices are here only used to build the preconditioner (see Section 3.4); if `JA(1)`< 0, diagonal preconditioning is required (see Section 4.5) and `B(1),...,B(N)` should contain the diagonal of B, whereas neither `JB` nor `IB` is accessed.

For `PJDREVCOM_GEP` and `JDREVCOM_GEP`, the reverse communication protocol is somehow more involved since, on return to the calling program, either a multiplication by A or by B or both may be required, and there is an additional related parameter `NDX3`; see comments in driver listings (Sections A.5 and A.6) for more details and the source file of EXAMPLE9_GEP for example of use.

Note that, for generalized eigenvalue problems, eigenvectors x_i returned in `X` are normalized in such a way that $x_i^* B\, x_i = 1$. However, the residual norm used for checking the convergence criterion is $\frac{\|Ax-\theta x\|}{\|Bx\|}$ (that is, JADAMILU tries to find $(\theta\,,\, x)$ such that $\frac{\|Ax-\theta x\|}{\|Bx\|} \leq$`TOL`). $\frac{\|Ax_i-\theta_i x_i\|}{\|Bx_i\|}$ this is also the quantity returned in vector `RES` and printed in program output.

Program output is similar to that for standard eigenvalue problems, but error estimates have a different meaning. For standard problems JADAMILU tries to deliver an error bound (i.e., a bound on the distance to closest exact eigenvalue), with some limitations explained in Section 6. For generalized eigenvalue problems, the same limitations apply and, in addition, error estimates are indicative in all cases. Indeed, a true bound should be based on $\frac{\|Ax_i-\theta x_i\|_{B^{-1}}}{\|Bx_i\|_{B^{-1}}}$, which cannot be computed since the inverse of B is not available. The printed error estimate uses therefore $\frac{\|Ax_i-\theta_i x_i\|}{\|Bx_i\|}$ as approximation to this quantity, and hence should be considered with care.

9 Complex matrices

Complex versions of JADAMILU drivers raise only few comments. All arguments remain real or integer except `A` and `X`, plus `B` in case of a generalized problem. Since A (B) is supposed Hermitian, it should have real diagonal entries, but this is not checked by JADAMILU. Note that `A` (`B`) has to be a complex array even when passing only the diagonal of A (B) for diagonal preconditioning.

See Appendix B for a further remark on matrix representation that is specific to the complex case.

A Drivers listings

A.1 DPJD

```
      SUBROUTINE DPJD( N, A, JA, IA, EIGS, RES, X, LX, NEIG,
     +                 SIGMA, ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +                 SHIFT, DROPTOL, MEM, ICNTL, IPRINT, INFO,
     +                 GAP)
      implicit none
*
*     .. Scalar Arguments ..
      integer           N, LX, NEIG, ISEARCH, NINIT, MADSPACE
      integer           ITER, ICNTL(5), IPRINT, INFO
      DOUBLE PRECISION            SIGMA, TOL, SHIFT, DROPTOL, MEM, GAP
*     ..
*     .. Array Arguments ..
      integer           JA(*), IA(*)
      DOUBLE PRECISION  A(*), X(*)
      DOUBLE PRECISION  EIGS(*), RES(*)
*     ..
*
*  Purpose
*  =======
*
*  Computes selected eigenvalues and eigenvectors of the symmetric matrix A
*  passed in arrays A, JA, IA, using built-in ILU preconditioning.
*
*  Arguments
*  =========
*
*  N        (input) INTEGER.
*           The dimension of the matrix. Should be larger than 1.
*
*  A        (input/output) numerical values
*  IA       (input/output) pointers for every column
*  JA       (input/output) column indices, if JA(1)<0, then A is assumed
*           to be diagonal.
*
*           Detailed description of A,IA,JA
*
*               Note first that no more than the diagonal part of A and
*               EITHER values from the strict upper triangular part OR
*               values from the strict lower triangular part are needed
*               to define the full matrix.
*               DPJD/DPJDREVCOM ASSUMES THAT EACH OFF-DIAGONAL ENTRY IS
*               REFERENCED ONLY ONCE, in either the upper triangular part
*               or in the lower triangular part.
*               (Thus, for instance, A,IA,JA may contain either the upper
*               triangular part or the lower triangular part - in each
*               case including diagonal).
*
*               DPJD ASSUMES IN ADDITION THAT ALL DIAGONAL ELEMENTS ARE
*               REFERENCED, EVEN WHEN SOME OF THEM ARE EQUAL TO ZERO.
*
```

```
*
*               OTHERWISE THE CODE WILL NOT RUN PROPERLY
*
*               On input, IA(I), I=1,...,N+1 refers to the physical start
*               of row I. In this case the entries of row I are located
*               in A(K), where K=IA(I),...IA(I+1)-1. JA(K) carries the
*               associated column indices. According what is written
*               above, DPJD assumes that some of these JA(K) (for
*               IA(I)<= K < IA(I+1) ) is equal to I with corresponding
*               A(K) carrying the value of the diagonal element, possibly
*               equal to zero.
*
*               A,IA,JA are "output" parameters because on exit the
*               entries of each row may occur in a different order (The
*               matrix is mathematically the same, but stored in
*               different way).
*
*  EIGS     (input/output) DOUBLE PRECISION array, dimension NEIG.
*           On input, eigenvalue estimates corresponding to provided
*               initial guesses (EIGS(i) corresponds to approximate
*               eigenvector number i); used only if NINIT>(MADSPACE+1)/2
*               to make sure that initial approximate eigenvectors are
*               processed in the right order.
*               Sorting is skipped if EIGS(1)=EIGS(2)= ... =EIGS(NINIT).
*               Then, if NINIT > (MADSPACE+1)/2 , initial approximations
*               should be in stored in increasing order of eigenvalues if
*               ISEARCH <= 1 , or in increasing distance of eigenvalues
*               to SIGMA if ISEARCH >= 2
*           On output, eigenvalues as they are computed
*               (EIGS(i) corresponds to approximate eigenvector number i).
*
*  RES      (output) DOUBLE PRECISION array, dimension NEIG.
*           Residual norms: RES(i)=|| A*x(i)-EIGS(i)*x(i) ||,
*               where A is the matrix, || is the two norm, and x(i) is
*               the approximate eigenvector number i .
*
*  X        (input/output+workspace) DOUBLE PRECISION array, dimension LX.
*           On input, the initial guess(es) (not required, see NINIT).
*           On output, the iterated approximate eigenvector(s).
*               On output (input), approximate eigenvector number i is
*               (or should be) stored in X(1+N*(i-1):N*i),
*               for i=1,...,NEIG (for i=1,...,NINIT)
*
*  LX       (input) INTEGER
*           Dimension of X. Should be at least
*               N*(2*MADSPACE+NEIG+4)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*           If MADSPACE >= 3, use LX not smaller than
*               N*(3*MADSPACE+NEIG+1)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*           to guarantee optimal performance.
*
*  NEIG     (input/output) INTEGER
*           On input, the number of eigenvalue(s) to be computed;
*                     should be positive.
*           On output, the number of eigenvalues effectively computed
*                     with the required accuracy.
```

```
*
*  SIGMA    (input) DOUBLE PRECISION
*           If ISEARCH <= 0: not used
*           If ISEARCH  = 1: estimation of the smallest eigenvalue
*                   (may speed up somewhat the algorithm if not too
*                    inaccurate)
*           If ISEARCH >= 2: the "target", see ISEARCH
*
*  ISEARCH (input) INTEGER
*           If ISEARCH <= 0: compute the smallest eigenvalue(s)
*           If ISEARCH  = 1: compute the smallest eigenvalue(s) and use
*                    SIGMA as initial guess. If one searches for the
*                    smallest eigenvalue(s) and has to rerun the
*                    algorithm for the same problem (or a problem with
*                    similar eigenvalues at the lower end), it is a good
*                    idea to set ISEARCH=1 and SIGMA=EIGS(1) (as obtained
*                    from the first run).
*           If ISEARCH >= 2: compute the eigenvalues closest to SIGMA
*
*  NINIT    (input) INTEGER
*           Number of initial guess(es) provided. May be set to 0.
*
*  MADSPACE (input) INTEGER
*            Maximal dimension of the search space (usually between 10
*            and 20). Should be at least 2.
*
*  ITER     (input/output) INTEGER
*           On input, the maximum number of matrix vector multiplications;
*                     should be positive.
*           On output, actual number of matrix vector multiplications.
*
*  TOL      (input) DOUBLE PRECISION
*           The tolerance on residual norm. Iterations to compute
*                eigenvector number i are stopped whenever
*                || A*x(i)-EIGS(i)*x(i) || <= TOL, where  ||x(i)||=1.
*                Should be positive.
*
*  SHIFT    (input/output) DOUBLE PRECISION
*           used only if ISEARCH = 1
*           On input, SHIFT is used to shift the input matrix by a
*               multiple of the identity matrix in order to compute the
*               preconditioner. A good heuristic is obtained by setting
*               SHIFT equal to SIGMA (the smallest eigenvalue estimate)
*               minus the estimated gap between this smallest eigenvalue
*               and the next one (i.e., SHIFT approximates 2 lambda_1 -
*               lambda_2, where lambda_1 (lambda_2) is smallest (second
*               smallest) eigenvalue). If one has no idea of this gap,
*               SHIFT may be set equal to SIGMA.
*           On output: suggested new value for the SHIFT parameter (not
*               necessarily equal to the current estimation of 2 lambda_1
*               - lambda_2). If one searches for the smallest eigenvalue(s)
*               and has to rerun the algorithm  for the same problem (or
*               a problem with similar eigenvalues at the lower end), it
*               is a good idea to set ISEARCH=1, SIGMA=EIGS(1) and SHIFT
*               equal to the value on output from DPJD/DPJDREVCOM.
```

```
*
*  DROPTOL (input/output) DOUBLE PRECISION
*          On input, drop tolerance for the multilevel incomplete
*              factorization. A small drop tolerance will typically lead
*              to more fill-in, i.e. more memory will be consumed and
*              the application of the preconditioner is more costly. On
*              the other hand, the number of iteration steps is expected
*              to be less for a smaller drop tolerance.
*          On output: suggested new value for the DROPTOL parameter,
*              that might be useful to if one has to rerun the algorithm
*              for a similar problem.
*
*  MEM     (input) DOUBLE PRECISION
*          MEM prescribes the amount of memory the user is willing to
*              spend for the preconditioner. MEM is relative to the
*              number of nonzero of the input matrix. If it turns out
*              the preconditioner that is computed does not fit into the
*              memory area that is offered by the user, DPJD will
*              terminate with an error message. In this case one can
*              either increase MEM (if there is more memory available)
*              or one has to increase DROPTOL.
*
*  ICNTL   (input) INTEGER
*          some control parameters
*          ICNTL(1) Not used (kept for compatibility with DPJDREVCOM).
*          ICNTL(2) If equal to zero, then adaptive preconditioning is
*                   used, i.e., during the eigenvector computation the
*                   ILU may be recomputed (with different SHIFT and
*                   DROPTOL), if useful and possible.
*                   If not equal to zero, then the preconditioner
*                   computed initially is kept throughout. If, in
*                   addition, ICNTL(2) is equal to -1, the existing
*                   preconditioner is reused in a static way (this
*                   option presumes that DPJD was called previously and
*                   successful for the same problem). Finally, ICNTL(2)
*                   set to -2 will cause that a previously existing
*                   preconditioner will be reused in a adaptive fashion.
*          ICNTL(3) If ICNTL(2) is equal to zero and ISEARCH <= 2,
*                   then ICNTL(3) states whether negative diagonal
*                   entries that show up in the ILU will be changed to
*                   positive ones. If set to zero (default), then up to
*                   1% of the negative diagonal entries are converted.
*                   If more negative diagonal are discovered then the
*                   algorithm searches for a new shift (and possibly a
*                   different DROPTOL if adaptive preconditioning is
*                   used).
*                   If ICNTL(3) is set to 1, then no negative diagonal
*                   entries are permitted forcing the algorithm to
*                   compute a different shift.
*          ICNTL(4) If set to zero, default estimate for norm of the
*                   inverse factor is used. Otherwise use ICNTL(4) as
*                   bound.
*          ICNTL(5) If ICNTL(5) is equal to 1, the preconditioner is
*                   computed with single precision arithmetic
*                   (all other computaion remain in double precision).
```

```
*
*  IPRINT  (input) INTEGER
*          Its absolute value indicates the unit number where
*              information is to be printed (N.B.: 5 is converted to 6).
*              If zero, only error messages are printed on standard
*              output. Otherwise, its sign indicates the level of output:
*              if negative, extensive information (for expert) is
*              provided; most users should be satisfied with the
*              information provided for positive IPRINT.
*
*  INFO    (output) INTEGER
*              INFO=0 if normal termination.
*              INFO>0 if allowed maximum number of matrix vector
*                        multplications performed without finding all
*                        wanted eigenvalues & eigenvectors.
*              INFO<0 if an error occurred - see printed output for
*                        details
*              (INFO=-54321: the computation of the preconditioner
*                        failed).
*
*  GAP     (output) DOUBLE PRECISION
*          The estimated distance between the set of NEIG computed
*          eigenvalues and the remaining part of the spectrum; may be
*          inaccurate.
*
```

A.2 DPJDREVCOM

```
      SUBROUTINE DPJDREVCOM( N, A, JA, IA, EIGS, RES, X, LX, NEIG,
     +                       SIGMA, ISEARCH, NINIT, MADSPACE, ITER,
     +                       TOL, SHIFT, DROPTOL, MEM, ICNTL,
     +                       IJOB, NDX1, NDX2, IPRINT, INFO, GAP)
      implicit none
*
*     .. Scalar Arguments ..
      integer            N, LX, NEIG, ISEARCH, NINIT, MADSPACE, INFO
      integer            ITER, ICNTL(5), IJOB, NDX1, NDX2, IPRINT
      DOUBLE PRECISION             SIGMA, TOL, SHIFT, DROPTOL, MEM, GAP
*     ..
*     .. Array Arguments ..
      integer            JA(*), IA(*)
      DOUBLE PRECISION             A(*), X(*)
      DOUBLE PRECISION             EIGS(*), RES(*)
*
*  Purpose
*  =======
*
*  Computes selected eigenvalues and eigenvectors of a symmetric matrix A
*  using built-in ILU or diagonal preconditioning.
*  Matrix multiplication by A is carried out in a user defined routine
*  that is called through a reverse communication protocol.
*
*  Arguments
*  =========
*
```

```
*     N, A, JA, IA,                          |  see comments in subroutine DPJD
*
*        In addition:
*
*             A, JA, IA need not to define exactly the matrix whose
*                eigenvalues are wanted; instead it may be some
*                approximation; this is consistent because the matrix
*                passed to DPJDREVCOM is only used to define a
*                preconditioner, whereas matrix vector multiplications
*                are performed by a user provided routine via the reverse
*                communication protocol (see below).
*
*             A, JA, IA should be compliant with the format described in DPJD
*             However:
*                   zero diagonal entries need not to be referenced in
*                         the structure (although they may);
*                   if a diagonal preconditioning is wanted, one
*                         should set JA(1) negative;
*                         if JA(1)<0, A(1),...,A(N) is supposed
*                         to carry the diagonal of the matrix;
*                         then, JA does not need to have a length greater
*                         than 1, IA is neither referenced, and A, JA, IA
*                         are unchanged on output.
*
*
*     EIGS, RES, X, LX, NEIG,                |
*     SIGMA, ISEARCH, NINIT, MADSPACE,       |
*     ITER, TOL, SHIFT, DROPTOL, MEM,        |  see comments in subroutine DPJD
*                                            |
*     IPRINT, INFO, GAP                      |
*
*
*  ICNTL    (input/output) INTEGER
*           some control parameters
*           ICNTL(1) should be set to zero (default value), except if X
*                    overwrites the arrays in A, JA, IA, in which case one
*                    should set ICNTL(1)=2 (this tells that the matrix
*                    cannot be refactored once the eigenvalue computation
*                    started)
*           other entries in ICNTL: see comments in subroutine DPJD
*
*
*  IJOB     (input/output) INTEGER.
*           Used to communicate job code between the levels.
*           Input: one should use IJOB=0 on the first call,
*                  and leave IJOB unchanged on subsequent calls
*           Output:
*                IJOB=0: work done - terminate
*                IJOB=1: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1)
*                        (call to matrix vector multiplication routine:
*                         MATVEC) and return to DPJDREVCOM leaving IJOB
*                        (and other parameters) unchanged.
*
*  NDX1     (output) INTEGER.
*  NDX2     Indicate indices into X() for the needed MATVEC when IJOB=1.
```

A.3 DJDREVCOM

```
*
      SUBROUTINE DJDREVCOM( N, EIGS, RES, X, LX, NEIG, SIGMA,
     +                      ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +                      IJOB, NDX1, NDX2, IPRINT, INFO, GAP)
      implicit none
*
*     .. Scalar Arguments ..
      integer            N, LX, NEIG, ISEARCH, NINIT, MADSPACE
      integer            ITER, IJOB, NDX1, NDX2, IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, GAP
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   X( * )
      DOUBLE PRECISION   EIGS( *), RES(*)
*
*  Purpose
*  =======
*
*  Computes selected eigenvalues and eigenvectors of a symmetric matrix A.
*  Matrix multiplication by A and preconditioner solve are carried out
*  in user defined routines that are called through a reverse
*  communication protocol.
*
*  Arguments
*  =========
*
*    N, EIGS, RES, X, LX, NEIG,          |
*    SIGMA, ISEARCH, NINIT,              |
*    MADSPACE, ITER, TOL,                |   see comments in subroutine DPJD
*                                        |
*    IPRINT, INFO, GAP                   |
*
*
*  IJOB     (input/output) INTEGER.
*           Used to communicate job code between the levels.
*           Input: one should use IJOB=0 on the first call, and leave
*                  IJOB unchanged on subsequent calls
*           Output:
*                IJOB=0: work done - terminate
*                IJOB=1: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1)
*                        (call to matrix vector multiplication routine:
*                         MATVEC) and return to DPJDREVCOM leaving IJOB
*                        (and other parameters) unchanged.
*                IJOB=2: solve Prec*X(NDX1:NDX1+N-1) = X(NDX2:NDX2+N-1)
*                        (call to preconditioner solve routine:
*                        PSOLVE) and return to DPJDREVCOM leaving IJOB
*                        (and other parameters) unchanged.
*
*  NDX1     (output) INTEGER.
*  NDX2     Indicate indices into X() for the needed MATVEC when IJOB=1,
*           or for the needed PSOLVE when IJOB=2.
*
```

A.4 DPJD_GEP

```
      SUBROUTINE DPJD_GEP( N, A, JA, IA, B, JB, IB, EIGS, RES, X, LX,
     +                     NEIG, SIGMA, ISEARCH, NINIT, MADSPACE, ITER,
     +                     TOL, SHIFT, DROPTOL, MEM, ICNTL, IPRINT,
     +                     INFO, GAP)
      implicit none
*
*     .. Scalar Arguments ..
      integer            N, LX, NEIG, ISEARCH, NINIT, MADSPACE
      integer            ITER, ICNTL(5), IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, SHIFT, DROPTOL, MEM, GAP
*     ..
*     .. Array Arguments ..
      integer            JA(*), IA(*), JB(*), IB(*)
      DOUBLE PRECISION   A(*), B(*), X(*)
      DOUBLE PRECISION   EIGS(*), RES(*)
*
*  Purpose
*  =======
*
*  Computes selected eigenvalues and eigenvectors of the generalized
*  eigenvalue problem defined by the symmetric matrix A and the symmetric
*  positive definite mass matrix B, passed in arrays A, JA, IA, and
*  arrays B, JB, IB, respectively; built-in ILU preconditioning is used.
*
*
*  Arguments
*  =========
*
*    N,A,JA,IA,                          |  see comments in subroutine DPJD
*
*  B, JB, IB (input/output) define the mass matrix B, using the same
*            storage scheme as for A (see comments in subroutine DPJD).
*            The same restriction on diagonal entries apply.
*            In addition, B should be positive definite.
*
*
*    EIGS, RES, X                       |
*                                       |
*    NEIG, SIGMA, ISEARCH, NINIT,       |
*    MADSPACE, ITER,                    |  see comments in subroutine DPJD
*                                       |
*    SHIFT, DROPTOL, MEM,               |
*    ICNTL, IPRINT, INFO, GAP           |
*
*
*  LX       (input) INTEGER
*           Dimension of X. If ISEARCH <= 1, should be at least
*              N*(3*MADSPACE+2*NEIG+4)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*           and, if ISEARCH >= 2, should be at least
*              N*(3*MADSPACE+2*NEIG)+MAX(4*N,6*MADSPACE)+3*MADSPACE**2
*                                   +MAX(MADSPACE**2,NEIG)
*           (which amounts the same except for very small N)
*           If MADSPACE >= 3, use LX not smaller than
```

```
*             N*(4*MADSPACE+2*NEIG+1)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*          to guarantee optimal performance.
*
*  TOL     (input) DOUBLE PRECISION
*          The tolerance on residual norm. Iterations to compute
*               eigenvector number i are stopped whenever
*               || A*x(i)-EIGS(i)*B*x(i) || <= TOL * ||B*x(i)||.
*               Should be positive.
*
```

A.5 DPJDREVCOM_GEP

```
      SUBROUTINE DPJDREVCOM_GEP( N, A, JA, IA, B, JB, IB, EIGS, RES, X,
     +                           LX, NEIG, SIGMA, ISEARCH, NINIT,
     +                           MADSPACE, ITER, TOL, SHIFT, DROPTOL,
     +                           MEM, ICNTL, IJOB, NDX1, NDX2, NDX3,
     +                           IPRINT, INFO, GAP)
      implicit none
*
*     .. Scalar Arguments ..
      integer    N, LX, NEIG, ISEARCH, NINIT, MADSPACE, ITER
      integer    ICNTL(5), IJOB, NDX1, NDX2, NDX3, IPRINT, INFO
      DOUBLE PRECISION      SIGMA, TOL, SHIFT, DROPTOL, MEM, GAP
*     ..
*     .. Array Arguments ..
      integer    JA(*), IA(*), JB(*), IB(*)
      DOUBLE PRECISION      A(*), B(*), X(*)
      DOUBLE PRECISION      EIGS(*), RES(*)
*
*  Purpose
*  =======
*
*  Computes selected eigenvalues and eigenvectors of the generalized
*  eigenvalue problem defined by a symmetric matrix A and a symmetric
*  positive definite mass matrix B, using a built-in ILU preconditioning.
*  Matrix multiplication by A and B is carried out in user defined
*   routine(s) that are called through a reverse communication protocol.
*
*
*  Arguments
*  =========
*
*    N,A,JA,IA,                          |    see comments in subroutine DPJD
*
*  B, JB, IB (input/output) define the mass matrix B, using the same
*            storage scheme as for A (see comments in subroutine DPJD).
*            The same restriction on diagonal entries apply.
*            In addition, B should be positive definite.
*
*       In addition:
*
*            A,JA,IA & B,JB,IB need not to define exactly the matrices
*               A and B; this is consistent because the matrix
*               passed to DPJDREVCOM_GEP is only used to define a
*               preconditioner, whereas matrix vector multiplications
```

```
*               are performed by a user provided routine via the reverse
*               communication protocol (see below).
*
*            A,JA,IA and B,JB,IB  should be compliant with the format
48 *            described in DPJD. However:
*                  zero diagonal entries need not to be referenced in
*                        the structure (although they may);
*                  if a diagonal preconditioning is wanted, one
*                        should set ja(1) negative;
*                        if JA(1)<0, A(1),...,A(N) is supposed
53 *                       to carry the diagonal of the matrix A, and
*                        B(1),...,B(N) the diagonal of the matrix B;
*                        then, JA does not need to have a length greater
*                        than 1, IA, IB, JB are neither referenced, and
58 *                       A,JA,IA & B,JB,IB are unchanged on output.
*
*
*    EIGS, RES, X,                          |
*                                           |
*    NEIG, SIGMA, ISEARCH, NINIT,           |
63 *    MADSPACE, ITER,                        |   see comments in subroutine DPJD
*                                           |
*    SHIFT, DROPTOL, MEM,                   |
*                                           |
*    IPRINT, INFO, GAP                      |
68 *
*
*  LX        (input) INTEGER
*            Dimension of X. If ISEARCH <= 1, should be at least
73 *              N*(3*MADSPACE+2*NEIG+4)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*            and, if ISEARCH >= 2, should be at least
*              N*(3*MADSPACE+2*NEIG)+MAX(4*N,6*MADSPACE)+3*MADSPACE**2
*                                    +MAX(MADSPACE**2,NEIG)
*            (which amounts the same except for very small N)
78 *            If MADSPACE >= 3, use LX not smaller than
*              N*(4*MADSPACE+2*NEIG+1)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*            to guarantee optimal performance.
*
*  TOL       (input) DOUBLE PRECISION
83 *            The tolerance on residual norm. Iterations to compute
*                 eigenvector number i are stopped whenever
*                 || A*x(i)-EIGS(i)*B*x(i) || <= TOL * ||B*x(i)||.
*                 Should be positive.
*
88 *  ICNTL     (input/output) INTEGER
*            some control parameters
*            ICNTL(1) should be set to zero (default value), except if X
*                     overwrites the arrays in A,JA,IA, in which case one
*                     should set ICNTL(1)=2 (this tells that the matrix
93 *                    cannot be refactored once the eigenvalue computation
*                     started)
*            other entries in ICNTL: see comments in subroutine DPJD
*
*  IJOB      (input/output) INTEGER.
98 *            Used to communicate job code between the levels.
```

```
*           Input: one should use IJOB=0 on the first call,
*                  and leave IJOB unchanged on subsequent calls
*           Output:
*                IJOB=0: work done - terminate
*                IJOB=1: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1)
*                        AND     X(NDX3:NDX3+N-1)= B*X(NDX1:NDX1+N-1)
*                        (call to matrix vector multiplication routine:
*                         MATVEC) and return to DPJDREVCOM_GEP leaving IJOB
*                        (and other parameters) unchanged.
*                IJOB=3: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1)
*                        (call to matrix vector multiplication routine:
*                         MATVEC) and return to DPJDREVCOM_GEP leaving IJOB
*                        (and other parameters) unchanged.
*                IJOB=4: compute X(NDX3:NDX3+N-1)= B*X(NDX1:NDX1+N-1)
*                        (call to matrix vector multiplication routine:
*                         MATVEC) and return to DPJDREVCOM_GEP leaving IJOB
*                        (and other parameters) unchanged.
*
*  NDX1     (output) INTEGER.
*  NDX2     Indicate indices into X() for the needed MATVEC when
*  NDX3     IJOB=1, 3 or 4.
*
```

A.6 DJDREVCOM_GEP

```
      SUBROUTINE DJDREVCOM_GEP(N, EIGS, RES, X, LX, NEIG, SIGMA,
     +                         ISEARCH, NINIT, MADSPACE, ITER, TOL,
     +                         IJOB, NDX1, NDX2, NDX3, IPRINT, INFO,
     +                         GAP)
      implicit none
*
*     .. Scalar Arguments ..
      integer            N, LX, NEIG, ISEARCH, NINIT, MADSPACE
      integer            ITER, IJOB, NDX1, NDX2, NDX3, IPRINT, INFO
      DOUBLE PRECISION   SIGMA, TOL, GAP
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   X( * )
      DOUBLE PRECISION   EIGS( *), RES(*)
*
*  Purpose
*  =======
*
*  Computes selected eigenvalues and eigenvectors of the generalized
*  eigenvalue problem defined by a symmetric matrix A and a symmetric
*  positive definite mass matrix B, using a built-in ILU preconditioning.
*  Matrix multiplication by A and B and preconditioner solve are
*  carried out in user defined routines that are called through a
*  reverse communication protocol.
*
*
*  Arguments
*  =========
*
*
```

```
*     EIGS, RES, X                          |
*                                           |
*     NEIG, SIGMA, ISEARCH, NINIT,          |
*     MADSPACE, ITER,                       |   see comments in subroutine DPJD
*                                           |
*     IPRINT, INFO, GAP                     |
*
*
*   LX        (input) INTEGER
*             Dimension of X. If ISEARCH <= 1, should be at least
*                N*(3*MADSPACE+2*NEIG+4)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*             and, if ISEARCH >= 2, should be at least
*                N*(3*MADSPACE+2*NEIG)+MAX(4*N,6*MADSPACE)+3*MADSPACE**2
*                                      +MAX(MADSPACE**2,NEIG)
*             (which amounts the same except for very small N)
*             If MADSPACE >= 3, use LX not smaller than
*                N*(4*MADSPACE+2*NEIG+1)+3*MADSPACE**2+MAX(MADSPACE**2,NEIG)
*             to guarantee optimal performance.
*
*   TOL       (input) DOUBLE PRECISION
*             The tolerance on residual norm. Iterations to compute
*                  eigenvector number i are stopped whenever
*                  || A*x(i)-EIGS(i)*B*x(i) || <= TOL * ||B*x(i)||.
*                  Should be positive.
*
*   IJOB      (input/output) INTEGER.
*             Used to communicate job code between the levels.
*             Input: one should use IJOB=0 on the first call, and leave
*                    IJOB unchanged on subsequent calls
*             Output:
*                  IJOB=1: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1)
*                          AND     X(NDX3:NDX3+N-1)= B*X(NDX1:NDX1+N-1)
*                          (call to matrix vector multiplication routine:
*                           MATVEC) and return to DPJDREVCOM_GEP leaving IJOB
*                          (and other parameters) unchanged.
*                  IJOB=2: solve Prec*X(NDX1:NDX1+N-1) = X(NDX2:NDX2+N-1)
*                          (call to preconditioner solve routine:
*                          PSOLVE) and return to DPJDREVCOM_GEP leaving IJOB
*                          (and other parameters) unchanged.
*                  IJOB=3: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1)
*                          (call to matrix vector multiplication routine:
*                           MATVEC) and return to DPJDREVCOM_GEP leaving IJOB
*                          (and other parameters) unchanged.
*                  IJOB=4: compute X(NDX3:NDX3+N-1)= B*X(NDX1:NDX1+N-1)
*                          (call to matrix vector multiplication routine:
*                           MATVEC) and return to DPJDREVCOM_GEP leaving IJOB
*                          (and other parameters) unchanged.
*
*   NDX1      (output) INTEGER.
*   NDX2      Indicate indices into X() for the needed MATVEC when
*   NDX3      IJOB=1, 3 or 4, or for the needed PSOLVE when IJOB=2.
*
```

A.7 PJDCLEANUP

```
      SUBROUTINE DPJDCLEANUP
*
*  Purpose
*  =======
*
*  Releases memory allocated by JADAMILU routines.
*
```

B Sparse matrix representation

We describe the representation of A (arrays `A`, `JA` and `IA`); in case of a generalized eigenvalue problem, B must be stored similarly in arrays arrays `B`, `JB` and `IB`.

Consider the symmetric matrix

$$A = \begin{pmatrix} 0 & 5 & 0 & 4 & 0 \\ 5 & 1 & 0 & 0 & -4 \\ 0 & 0 & 0 & 5 & 0 \\ 4 & 0 & 5 & 1 & 0 \\ 0 & -4 & 0 & 0 & 0 \end{pmatrix} \in \mathbb{R}^{5,5},$$

In order to interact with PJD or PJDREVCOM only half of the matrix should be passed, i.e. you can either pass the diagonal part plus the strict upper triangular part or the diagonal part plus the strict lower triangular part. You may also pass any mixture of both; e.g., a symmetric permutation of the the upper triangular part. In all cases, you should avoid duplicated entries: if an entry from the lower triangular part is passed, the associated upper triangular part must not be passed. On the other hand, all diagonal entries must be explicitly passed, even when they are equal to zero. These two requirements should be met: **otherwise the code will not run properly**.

The following examples are legal possibilities, missing entries are those that will not be passed.

$$\begin{pmatrix} 0 & 5 & & 4 & \\ & 1 & & & -4 \\ & & 0 & 5 & \\ & & & 1 & \\ & & & & 0 \end{pmatrix}, \begin{pmatrix} 0 & & & & \\ 5 & 1 & & & \\ & & 0 & & \\ 4 & & 5 & 1 & \\ & -4 & & & 0 \end{pmatrix}, \begin{pmatrix} 0 & & & & \\ 5 & 1 & & & -4 \\ & & 0 & & \\ 4 & & 5 & 1 & \\ & & & & 0 \end{pmatrix}.$$

Note that in the complex case the matrix is Hermitian instead of simply symmetric. Again only half of the offdiagonal entries should be passed, and this should be interpreted as follows: if, say $A_{34} = 2 + 3i$ and hence $A_{43} = 2 - 3i$, you should either pass $2 + 3i$ as entry in position $(3, 4)$ or $2 - 3i$ as entry in position $(4, 3)$. On the other hand, diagonal entries must be real but nevertheless stored as complex numbers.

The data structures that are required by JADAMILU are based on the so–called compressed sparse row format (CSR).
We kindly refer to the SPARSKIT package `http://www-users.cs.umn.edu/~saad` for some tools to convert various sparse matrix formats to CSR format.

For every row only the nonzero entries need to be stored, except for the diagonal entries that always have to be stored. The rows are stored one after another. To do that there are three arrays. The first array is used as pointer array to separate the single rows, i.e. if the first row has three nonzero entries and the second row has two nonzero entries, then the pointer array starts with $1, 4, 6, \ldots$. If you have an $N \times N$ matrix, then this pointer array needs $N + 1$ entries. The last entry is used for compatibility and indicates the first spot behind the matrix.

Consider the following matrix

$$\begin{pmatrix} 0 & 5 & & 4 & \\ & 1 & & & -4 \\ & & 0 & 5 & \\ & & & 1 & \\ & & & & 0 \end{pmatrix} \qquad \begin{matrix} \text{pointer } \texttt{IA} \\ 1 \\ 4 \\ 6 \\ 8 \\ 9 \\ 10 \end{matrix}$$

Next there are two arrays that hold the column indices of the nonzero and diagonal entries as well as their numerical values

$$\begin{pmatrix} 0 & 5 & & 4 & \\ & 1 & & & -4 \\ & & 0 & 5 & \\ & & & 1 & \\ & & & & 0 \end{pmatrix} \qquad \begin{matrix} \text{pointer } \texttt{IA} \\ 1 \\ 4 \\ 6 \\ 8 \\ 9 \\ 10 \end{matrix} \qquad \begin{matrix} \text{index } \texttt{JA} \\ 1 \quad 2 \quad 4 \\ 2 \quad 5 \\ 3 \quad 4 \\ 4 \\ 5 \end{matrix} \qquad \begin{matrix} \text{values } \texttt{A} \\ 0.0 \quad 5.0 \quad 4.0 \\ 1.0 \quad -4.0 \\ 0.0 \quad 5.0 \\ 1.0 \\ 0.0 \end{matrix}$$

Altogether we have to pass the following arrays

$$\begin{aligned} \texttt{IA} &= \left[\begin{array}{cccccc} 1 & 4 & 6 & 8 & 9 & 10 \end{array}\right] \\ \texttt{JA} &= \left[\begin{array}{ccc|cc|cc|c|c} 1 & 2 & 4 & 2 & 5 & 3 & 4 & 4 & 5 \end{array}\right] \\ \texttt{A} &= \left[\begin{array}{ccc|cc|cc|c|c} 0.0 & 5.0 & 4.0 & 1.0 & -4.0 & 0.0 & 5.0 & 1.0 & 0.0 \end{array}\right] \end{aligned}$$

C JADAMILU parameters in alphabetical order

A

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input/output) REAL / DOUBLE PRECISION / COMPLEX / DOUBLE COMPLEX array
On input, entries of matrix A, see Appendix B for a detailed description.
PJDREVCOM and PJDREVCOM_GEP only: if JA(1)< 0, diagonal preconditioning is wanted and A(1),...,A(N) should carry the diagonal of A.
A is an output parameter because on exit the entries of each row may occur in a different order (The matrix is mathematically the same, but stored in different way).

B

Only used in PJD_GEP and PJDREVCOM_GEP
(input/output) REAL / DOUBLE PRECISION / COMPLEX / DOUBLE COMPLEX array
On input, entries of matrix B, see Appendix B for a detailed description.
PJDREVCOM_GEP only: if JA(1)< 0, diagonal preconditioning is wanted and B(1),...,B(N) should carry the diagonal of B.
B is an output parameter because on exit the entries of each row may occur in a different order (The matrix is mathematically the same, but stored in different way).

DROPTOL

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input/output) REAL / DOUBLE PRECISION
On input, drop tolerance for the multilevel incomplete factorization. A small drop tolerance will typically lead to more fill-in, i.e. more memory will be consumed and the application of the preconditioner is more costly. On the other hand, the number of iteration steps is expected to be less for a smaller drop tolerance.
On output: suggested new value for the DROPTOL parameter, that might be useful to if one has to rerun the algorithm for a similar problem.

EIGS

(input/output) REAL / DOUBLE PRECISION array, dimension NEIG
On input, eigenvalue estimates corresponding to provided initial guesses (EIGS(i) corresponds to approximate eigenvector number i; used only if NINIT$>$(MADSPACE+1)/2 to make sure that initial approximate eigenvectors are processed in the right order. Sorting is skipped if EIGS(1)=EIGS(2)=...=EIGS(NINIT). Then, if NINIT$>$(MADSPACE+1)/2 , initial approximations should be in stored in increasing order of eigenvalues if ISEARCH≤ 1, or in increasing distance of eigenvalues to SIGMA if ISEARCH≥ 2.
On output, eigenvalues as they are computed (EIGS(i) corresponds to approximate eigenvector number i).

GAP

(output) REAL / DOUBLE PRECISION
The estimated distance between the set of NEIG computed eigenvalues and the remaining part of the spectrum; may be inaccurate.

IA

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input) INTEGER array, dimension N+1
Pointer for rows in matrix A, see Appendix B for a detailed description.
PJDREVCOM and PJDREVCOM_GEP only: not accessed if if JA(1)< 0 (diagonal preconditioning).

IB

Only used in PJD_GEP and PJDREVCOM_GEP
(input) INTEGER array, dimension N+1
Pointer for rows in matrix B, see Appendix B for a detailed description.
PJDREVCOM_GEP only: not accessed if if JA(1)< 0 (diagonal preconditioning).

ICNTL

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input) INTEGER array, dimension 5.
Some control parameters:

ICNTL(1) Only used in PJDREVCOM and PJDREVCOM_GEP .
ICNTL(1) should be set to zero (default value), except if X overwrites the arrays in A,JA,IA, in which case one should set ICNTL(1)$= 2$ (this tells that the matrix cannot be re-factored once the eigenvalue computation started).

ICNTL(2) If equal to zero, then adaptive preconditioning is used, i.e., during the eigenvector computation the ILU may be recomputed (with different SHIFT and DROPTOL), if useful and possible. If not equal to zero, then the preconditioner computed initially is kept throughout. If, in addition, ICNTL(2)$= -1$, the existing preconditioner is reused in a static way (this option presumes that PJD was called previously and successful for the same problem). Finally, ICNTL(2) set to -2 will cause that a previously existing preconditioner will be reused in a adaptive fashion.

ICNTL(3) If ICNTL(2) is equal to zero and ISEARCH$\neq 2$, then ICNTL(3) states whether negative diagonal entries that show up in the ILU will be changed to positive ones. If set to zero (default), then up to 1% of the negative diagonal entries are converted. If more negative diagonal are discovered then the algorithm searches for a new shift (and possibly a different DROPTOL if adaptive preconditioning is used).
If ICNTL(3) is set to 1, then no negative diagonal entries are permitted forcing the algorithm to compute a different shift.

ICNTL(4) If set to zero, default estimate for norm of the inverse factor is used. Otherwise use ICNTL(4) as bound.

ICNTL(5) Ignored by single precision versions.
For double precision versions, when ICNTL(5)= 1, then the preconditioner is computed with single precision arithmetic (other computation remain in double precision).

IJOB

Only used in PJDREVCOM, JDREVCOM, PJDREVCOM_GEP and JDREVCOM_GEP
(input/output) INTEGER
Used to control reverse communication protocol.

Input: one should use IJOB=0 on the first call, and leave IJOB unchanged on subsequent calls

Output:
- IJOB=0: work done – terminate
- IJOB=1: compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1) (call to matrix vector multiplication routine) and return to PJDREVCOM leaving IJOB (and other parameters) unchanged.
  PJDREVCOM_GEP and JDREVCOM_GEP only: in addition, compute also X(NDX3:NDX3+N-1)= B*X(NDX1:NDX1+N-1) before returning.
- IJOB=2: (JDREVCOM and JDREVCOM_GEP only)
  solve M*X(NDX1:NDX1+N-1)=X(NDX2:NDX2+N-1) (call to preconditioner solve routine) and return to PJDREVCOM leaving IJOB (and other parameters) unchanged.
- IJOB=3: (PJDREVCOM_GEP and JDREVCOM_GEP only) compute X(NDX2:NDX2+N-1)= A*X(NDX1:NDX1+N-1) (call to matrix vector multiplication routine) and return to PJDREVCOM leaving IJOB (and other parameters) unchanged.
- IJOB=4: (PJDREVCOM_GEP and JDREVCOM_GEP only) compute X(NDX3:NDX3+N-1)= B*X(NDX1:NDX1+N-1) (call to matrix vector multiplication routine) and return to PJDREVCOM leaving IJOB (and other parameters) unchanged.

INFO

(output) INTEGER

- INFO= 0 if normal termination.
- INFO> 0 if allowed maximum number of matrix vector multiplications performed without finding all wanted eigenvalues & eigenvectors.
- INFO< 0 if an error occurred - see printed output for details
- INFO= −54321 (PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP only)
  the computation of the preconditioner failed.

IPRINT

(input) INTEGER
Its absolute value indicates the unit number where information is to be printed (N.B.: 5 is converted to 6). If zero, only error messages are printed on standard output. Otherwise, its

sign indicates the level of output: if negative, extensive information (for expert) is provided; most users should be satisfied with the information provided for positive IPRINT.

ISEARCH

(input) INTEGER

ISEARCH≤ 0	compute the smallest eigenvalue(s)
ISEARCH$= 1$	compute the smallest eigenvalue(s) and use SIGMA as initial guess. If one searches for the smallest eigenvalue(s) and has to rerun the algorithm for the same problem (or a problem with similar eigenvalues at the lower end), it is a good idea to set ISEARCH$= 1$ and SIGMA=EIGS(1) (as obtained from the first run).
ISEARCH$= 2$	compute the eigenvalues closest to SIGMA

ITER

(input/output) INTEGER
On input, the maximum number of matrix vector multiplications; should be positive.
On output, actual number of matrix vector multiplications.

JA

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input/output) INTEGER array
Column indices of entries in matrix A, see Appendix B for a detailed description.
PJDREVCOM and PJDREVCOM_GEP only: if JA(1)< 0, further entries are not accessed (diagonal preconditioning).
JA is an output parameter because on exit the entries of each row may occur in a different order (The matrix is mathematically the same, but stored in different way).

JB

Only used in PJD_GEP and PJDREVCOM_GEP
(input/output) INTEGER array
Column indices of entries in matrix B, see Appendix B for a detailed description.
PJDREVCOM_GEP only: not accessed if if JA(1)< 0 (diagonal preconditioning).
JB is an output parameter because on exit the entries of each row may occur in a different order (The matrix is mathematically the same, but stored in different way).

LX

(input) INTEGER
Dimension of X.

PJD, PJDREVCOM and JDREVCOM: should be at least

$$\mathtt{N}(2\,\mathtt{MADSPACE} + \mathtt{NEIG} + 4) + 3\,\mathtt{MADSPACE}^2 + \max(\mathtt{MADSPACE}^2, \mathtt{NEIG}) \ ;$$

if MADSPACE≥ 3, use LX not smaller than

$$\texttt{N}(3\,\texttt{MADSPACE}+\texttt{NEIG}+1)+3\,\texttt{MADSPACE}^2+\max(\texttt{MADSPACE}^2,\texttt{NEIG})$$

to guarantee optimal performance.

PJD_GEP, PJDREVCOM_GEP and JDREVCOM_GEP: should be at least

$$\texttt{N}(3\,\texttt{MADSPACE}+2\,\texttt{NEIG}+4)+3\,\texttt{MADSPACE}^2+\max(\texttt{MADSPACE}^2,\texttt{NEIG})\ ;$$

if ISEARCH≥ 2, should be also not less than

$$\texttt{N}(3\,\texttt{MADSPACE}+2\,\texttt{NEIG})+6\,\texttt{MADSPACE}+3\,\texttt{MADSPACE}^2+\max(\texttt{MADSPACE}^2,\texttt{NEIG})\ ;$$

if MADSPACE≥ 3, use LX not smaller than

$$\texttt{N}(4\,\texttt{MADSPACE}+2\,\texttt{NEIG}+1)+3\,\texttt{MADSPACE}^2+\max(\texttt{MADSPACE}^2,\texttt{NEIG})$$

to guarantee optimal performance.

MADSPACE

(input) INTEGER
Maximal dimension of the search space (usually between 10 and 20). Should be at least 2.

MEM

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
MEM prescribes the amount of memory the user is willing to spend for the preconditioner. MEM is relative to the number of nonzero of the input matrix. If it turns out the preconditioner that is computed does not fit into the memory area that is offered by the user, PJDINIT will terminate with an error message. In this case one can either increase MEM (if there is more memory available) or one has to increase DROPTOL.

N

(input) INTEGER.
The dimension of the matrix. Should be larger than 1.

NDX1

Only used in PJDREVCOM, JDREVCOM, PJDREVCOM_GEP and JDREVCOM_GEP
(output) INTEGER
See description of IJOB.

NDX2

Only used in PJDREVCOM, JDREVCOM, PJDREVCOM_GEP and JDREVCOM_GEP
(output) INTEGER
See description of IJOB.

NDX3

Only used in PJDREVCOM_GEP and JDREVCOM_GEP
(output) INTEGER
See description of IJOB.

NEIG

(input/output) INTEGER
On input, the number of eigenvalue(s) to be computed; should be positive.
On output, the number of eigenvalues effectively computed with the required accuracy.

NINIT

(input) INTEGER
Number of initial guess(es) provided. May be set to 0.

RES

(output) REAL / DOUBLE PRECISION array, dimension NEIG.
Residual norms.

PJD, PJDREVCOM and JDREVCOM:
RES(i) $= \|Ax_i - \texttt{EIGS(i)}x_i\|$, where A is the matrix, $\|$ is the two norm, and x_i is the approximate eigenvector number i stored in X(1+N*(i-1):N*i) (normalized such that $\|x_i\| = 1$).

PJD_GEP, PJDREVCOM_GEP and JDREVCOM_GEP: RES(i) $= \frac{\|Ax_i - \texttt{EIGS(i)}Bx_i\|}{\|Bx_i\|}$, where A is the matrix, B is the mass matrix, $\|$ is the two norm, and x_i is the approximate eigenvector number i stored in X(1+N*(i-1):N*i).

SHIFT

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input/output) REAL / DOUBLE PRECISION
On input (ISEARCH= 1), SHIFT is used to shift the input matrix by a multiple of the identity matrix in order to compute the preconditioner. A good heuristic is obtained by setting SHIFT equal to SIGMA (the smallest eigenvalue estimate) minus the estimated gap between this smallest eigenvalue and the next one (i.e., SHIFT approximates $2\lambda_1 - \lambda_2$, where λ_1 (λ_2) is smallest (second smallest) eigenvalue). If one has no idea of this gap, SHIFT may be set equal to SIGMA.
On output (ISEARCH$\leq$ 1): suggested new value for the SHIFT parameter (not necessarily equal to the current estimation of $2\lambda_1 - \lambda_2$). If one searches for the smallest eigenvalue(s) and has to rerun the algorithm for the same problem (or a problem with similar eigenvalues at the lower end), it is a good idea to set ISEARCH= 1, SIGMA=EIGS(1) and SHIFT equal to the value on output from PJD/PJDREVCOM.

SIGMA

Only used in PJD, PJDREVCOM, PJD_GEP and PJDREVCOM_GEP
(input) REAL / DOUBLE PRECISION
ISEARCH≤ 0 not used
ISEARCH$= 1$ estimation of the smallest eigenvalue (may speed up somewhat the algorithm if not too inaccurate)
ISEARCH≥ 2 the "target", see ISEARCH

TOL

(input) DOUBLE PRECISION
The tolerance on residual norm. Should be positive.

PJD, PJDREVCOM and JDREVCOM:
Iterations to compute eigenvector number i are stopped whenever $\|Ax_i - \texttt{EIGS(i)}x_i\| \leq \texttt{TOL}$ (with $\|x_i\| = 1$).

PJD_GEP, PJDREVCOM_GEP and JDREVCOM_GEP: Iterations to compute eigenvector number i are stopped whenever $\|Ax_i - \texttt{EIGS(i)}Bx_i\| \leq \texttt{TOL}\ \|Bx_i\|$.

X

(input/output+workspace) REAL / DOUBLE PRECISION / COMPLEX / DOUBLE COMPLEX array, dimension LX.
On input, the initial guess(es) (not required, see NINIT).
On output, the iterated approximate eigenvector(s).
On output (input), approximate eigenvector number i is (or should be) stored in X(1+i-1):N*i), for i=1,...,NINIT.

D Acknowledgments and Copyrights

Using JADAMILU is free for non commercial applications (For commercial use, please contact the authors). You can acknowledge, using reference [4], the contribution of this package in any scientific publication dependent upon the use of the package. You shall use reasonable endeavors to notify the authors of the package of this publication.

Note that JADAMILU also contains code that is not owned by the authors. If you plan to use it in different way please contact the authors. In particular, JADAMILU uses BLAS, LAPACK, MC64 and AMD.

D.1 MC64

We kindly thank Prof. Iain S. Duff for the permission to add MC64 to this library. We refer to [11] if the user is interested in the mathematical background. Please note the following copyright.

```
C COPYRIGHT (c) 1999 Council for the Central Laboratory
*                    of the Research Councils
CCCCC PACKAGE MC64A/AD
CCCCC AUTHORS Iain Duff (i.duff@rl.ac.uk) and Jacko Koster (jak@ii.uib.no)
CCCCC LAST UPDATE 20/09/99
CCCCC
C *** Conditions on external use ***
C
C The user shall acknowledge the contribution of this
C package in any publication of material dependent upon the use of
C the package. The user shall use reasonable endeavours to notify
C the authors of the package of this publication.
C
C The user can modify this code but, at no time
C shall the right or title to all or any part of this package pass
C to the user. The user shall make available free of charge
C to the authors for any purpose all information relating to any
C alteration or addition made to this package for the purposes of
C extending the capabilities or enhancing the performance of this
C package.
C
C The user shall not pass this code directly to a third party without
C the express prior consent of the authors.  Users wanting to licence
C their own copy of these routines should send email to hsl@aeat.co.uk
C
C None of the comments from the Copyright notice up to and including
C this one shall be removed or altered in any way.
```

D.2 AMD

AMD is a set of routines for ordering a sparse matrix prior to Choleski factorization (or for LU factorization with diagonal pivoting).

Copyright (c) 2004-2006 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. All

Rights Reserved. Distributed under the GNU LGPL license.

See e.g. [1] for a description of this method.

D.3 BLAS

The reference BLAS is a freely-available software package. It is available from netlib via anonymous ftp and the World Wide Web. Thus, it can be included in commercial software packages (and has been). We only ask that proper credit be given to the authors.

Like all software, it is copyrighted. It is not trademarked, but we do ask the following:

If you modify the source for these routines we ask that you change the name of the routine and comment the changes made to the original.

We will gladly answer any questions regarding the software. If a modification is done, however, it is the responsibility of the person who modified the routine to provide support.

Publications/references for the BLAS, see [12, 10, 9, 8, 7], new BLAS see [3, 6].

D.4 LAPACK

The complete LAPACK package or individual routines from LAPACK are freely available on netlib and can be obtained via the World Wide Web or anonymous ftp.

The LAPACK homepage can be accessed on the World Wide Web via the URL address:

`http://www.netlib.org/lapack/`

As a standard reference see [2].

References

[1] P. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, , and D. Sorensen, *LAPACK Users' Guide, Third Edition*, SIAM Publications, 1999.

[3] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, *An updated set of Basic Linear Algebra Subprograms (BLAS)*, ACM Trans. Math. Software, 28 (2002), pp. 135–151.

[4] M. Bollhöfer and Y. Notay, *JADAMILU: a software code for computing selected eigenvalues of large sparse symmetric matrices*, Computer Physics Communications, 177 (2007), pp. 951–964.

[5] M. Bollhöfer, Y. Saad, and O. Schenk, *ILUPACK — preconditioning software package, volume 2.1*, January 2006.

[6] J. Dongarra, *Basic Linear Algebra Subprograms technical forum standard*, Int. J. of High Perf. Appl. and Supercomp., 16 (2002), pp. 1–199.

[7] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, *Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 18–28.

[8] ——, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[9] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 18–32.

[10] ——, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.

[11] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 889–901.

[12] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–323.

[13] G. Sleijpen and H. A. van der Vorst, *A Jacobi–Davidson iteration method for linear eigenvalue problems*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 401–425.

[14] ——, *A Jacobi–Davidson iteration method for linear eigenvalue problems*, SIAM Review, 42 (2000), pp. 267–293.