Reverse Communication Interface for Linear Algebra Templates for Iterative Methods

Jack Dongarra
University of Tennessee and Oak Ridge National Laboratory
Victor Eijkhout
University of California, Los Angeles
Ajay Kalhan
University of Tennessee

May 18, 1995

1 Introduction

In this report we describe a *reverse communication interface* for the software implementing the iterative methods described in the *Templates book* [2]. Reverse communication is a technique by which we can hide the implementation details of various operations from the implementation of the iterative method. This allows us to (a) remove references to the user-prepared array or data structure containing the matrix within the iterative solver, and (b) uniformly take care of the various components that can be changed by a user. These include implementation details of matrix-vector operations, vector operations, stopping tests, and norm computations.

Section 2 discusses the concept of reverse communication with the help of an example from the Templates code. Section 3 explains the use of iterative methods under this scheme. Section 4 discusses refinements and enhancements that we have planned. Finally, Section 5 provides information on how to obtain the software and the Appendix presents an example code.

2 Reverse Communication

In this section, we present the motivation behind reverse communication. We then describe how to use the interface and discuss its advantages and limitations.

2.1 The Interface

All iterative methods have one or more of the following operations:

- matrix-vector multiply (transpose and/or nontranspose), and
- preconditioner solve (transpose and/or nontranspose).

These operations account for a large portion of the computation and thus are the focus of most optimization efforts. Optimization can involve significant changes in algorithm and data structures. Further, different problems may require different storage formats for their associated matrices and vectors.

The primary aim of reverse communication (REVCOM) is to isolate the matrix-vector operation. Such operations are performed by routines that the user supplies on data structures that are most natural to the problem at hand [3]. The idea of reverse communication is not new. In this approach, the iterative method subroutine does not need to know anything about the data structure or other requirements of the matrix-vector or preconditioner operations. Whenever the iterative method subroutine needs the results of one of the operations, it returns control to the user's subroutine that called it. The user's subroutine then invokes the module that performs the operation. The iterative method subroutine is invoked again by the user with the results of the operation. This interaction is explained in Figure 1 with the help of the conjugate gradient (CG) iterative method.

Here, CGREVCOM implements the iterative method, and CG implements the user subroutine that manages the requests from CGREVCOM. The user's main program invokes CG as before. The CG subroutine invokes CGREVCOM; and during this call, the CG method and *resume logic* are initialized. One step of the initialization of CG is a matrix-vector multiply. At this point in the flow of control, the job indicator variable (IJOB) is set to 1, and control is returned to CG. In CG, control resumes at the case statement. When IJOB = 1, the MATVEC subroutine is invoked. On return from MATVEC, control returns to the line with label 10. CGREVCOM is then invoked, this time with the results of the matrix-vector multiply. Inside CGREVCOM, the *resume logic* is used to determine where execution should resume. A similar sequence of actions occurs within the ITERATIVE METHOD MAIN LOOP when results of the preconditioner solve and matrix-vector multiply are required. For the former, CG invokes PSOLVE; for the latter, it invokes MATVEC.

Data is exchanged by using the following parameters of CGREVCOM.

1. *IJOB:* used by CGREVCOM to tell CG of the operation it needs done, and used by CG to tell CGREVCOM whether the current invocation is an initialization call or a resume call.
2. *NDX1, NDX2:* used by CGREVCOM to tell CG of the indices of the operand and to identify the destination vectors to be used in the operations. This is for the cases when these vectors are located in the WORK array. If either is a named

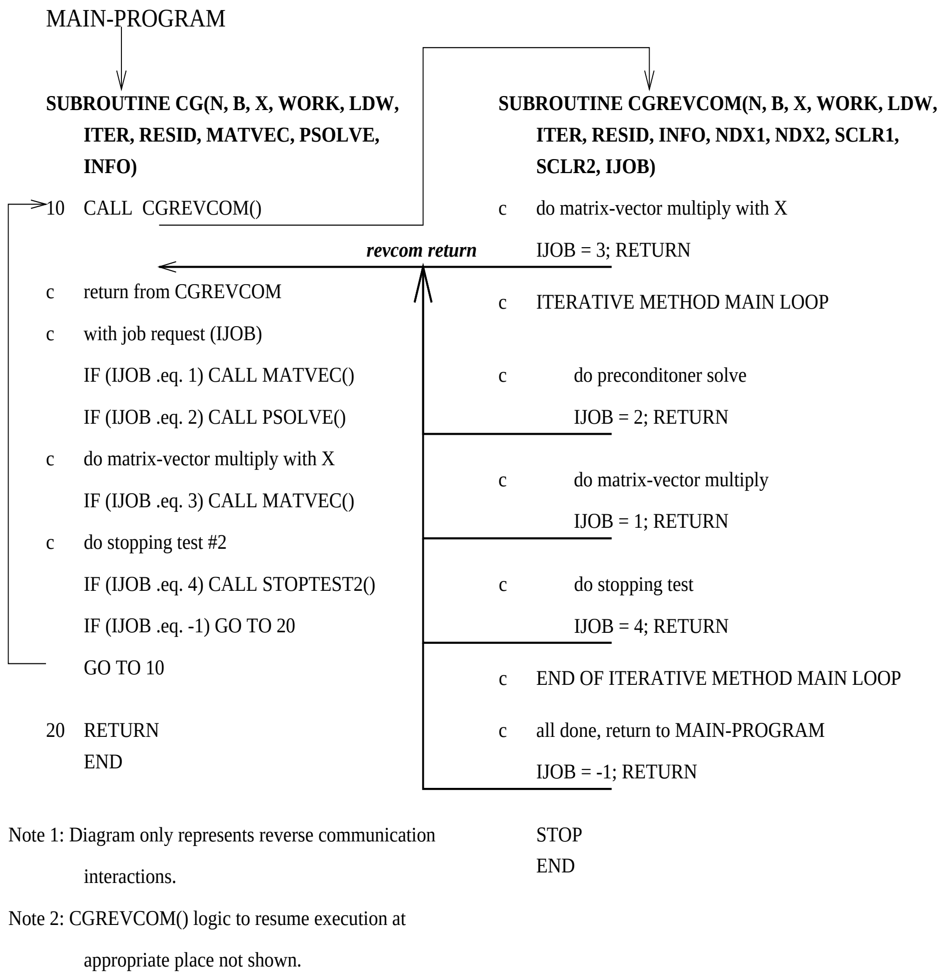

Figure 1: Interactions as a result of reverse communication

vector (e.g., X), a different opcode is used to indicate that the operation has to be done by using vector X.

3. *SCLR1, SCLR2:* used by CGREVCOM to tell CG what the scalars in the matrix-vector multiplication are. The Templates code currently uses *dgemm* from LAPACK, and that routine requires two scalar arguments—*alpha* and *beta*.

Note that we have implemented reverse communication only for matrix-vector operations. Vector-vector operations (dot product, norm computation) operate on contiguous memory locations, and there is not any variation in the implementation of such operations. However, if required, these vector-vector operations can also be easily provided in the reverse communication interface.

2.2 Another Use of the Interface

REVCOM provides a *client-server* type of interface between the iterative method routine and the interface. That is, the reverse communication asks for services (i.e., is the client), and the interface level provides the requested services (i.e., the server). This model allows us to abstract from the iterative method routine all pieces of code that are liable to change as a result of user requirements. An important example is the above discussion on moving the matrix-vector operations out of the iterative method routine.

In the same vein, we can abstract the stopping test, too. It is generally accepted that there is not one best stopping test for a given combination of user problem and iterative method. Thus, we can use the REVCOM interface to allow the user to change stopping tests quite cleanly. Without such an interface, all changes would have to be made directly in the iterative method routine. Since this routine can be complicated, there is an increased risk of introducing collateral errors.

The stopping test was included in the interface by making the following modifications:

- A job code was created with which the client can request the stopping test to be performed.

- Different stopping tests may operate on different vectors. Further, only the client routine knows the layout of the vectors. Therefore, the server, depending on the stopping test, has to tell the client which vector's indices it needs. This step is taken before the *init* call to the client routine, by setting the NDX1 and NDX2 parameters to appropriate *code*s. These codes (one for each vector) are defined by the client and copied by the server. This mechanism is possible because the workspace, WORK, can also be referenced in the server routine.
  The server performs the stopping test and conveys the result to the client via the INFO parameter. A value of 1 indicates a successful test.

- The client, during the *init* call, records the indices needed by the server in local variables NEED1 and NEED2. Subsequently, every time it requests the stopping test to be performed, it sets output parameters NDX1 and NDX2 to NEED1 and

```
      SUBROUTINE CG( )

c     Codes for CGREVCOM's vectors.
c     1 == R; 2 == Z; 3 == P; 4 == Q;
c     -1 == ignore; any other == error

      NDX1 = 1
      NDX2 = -1

      IJOB = 1

10    CALL  CGREVCOM( )
        .
        .
        .
```

```
      SUBROUTINE CGREVCOM( )

c     Alias workspace columns.
      R = 1; Z = 2; P = 3; Q = 4
      IF( NDX1.NE.-1 ) THEN
        IF( NDX1.EQ.1 ) THEN NEED1 = ((R - 1) * LDW) + 1
        ELSEIF( NDX1.EQ.2 ) THEN NEED1 = ((Z - 1) * LDW) + 1
        ELSEIF( NDX1.EQ.3 ) THEN NEED1 = ((P - 1) * LDW) + 1
        ELSEIF( NDX1.EQ.4 ) THEN NEED1 = ((Q - 1) * LDW) + 1
      ENDIF
c     Similar code for NDX2/NEED2
        .
        .
        .
c     Request stopping test
        NDX1 = NEED1
        NDX2 = NEED2
        RLBL = 5
        IJOB = 4
        RETURN
        .
        .
        .
```

Figure 2: Actions for REVCOM stopping test

NEED2, respectively. For the CG algorithm, with stopping criterion 2 from the Templates book (i.e., $||r^{(i)}|| \leq stop\_tol\ ||b||$), the actions are shown in Figure 2. In the client routine, the code for the stopping test is replaced by code for setting the job code in IJOB, resume label in RLBL, and control returned to the server. When the client is reinvoked after the stopping test, it tests INFO to check whether the stopping test was passed.

2.3 Comments on the Interface

The subroutine (CG in Figure 1) that implements the reverse communication interface must be provided by the user. Apart from any other code, it must contain the GOTO and the IF statements, since these implement the reverse communication feature. Users can use CG and corresponding routines for other iterative methods as templates for their own code.

As mentioned above, the purpose of the REVCOM interface is to isolate the iterative method subroutine from the implementation details of certain operations. A side effect is the slightly unnatural programming style that it imposes. In standard Fortran 77, comparable functionality can be achieved by (a) using COMMON blocks, or (b) passing user subroutines to the iterative method subroutine. COMMON blocks allow for global variables that, for software engineering reasons, are not the preferred means for sharing data. Further, since multiple processes might read/write the same block, memory consistency requirements discourage the use of COMMON blocks. With option (b) one drawback is that we do not get the same level of isolation as with REVCOM. Another is that the calling sequence has to be frozen[1]. In comparison, the REVCOM interface provides us with a rational framework for incorporating different matrix storage formats and different implementations of the operations.

Programs written with the REVCOM framework are as clean, easy to develop, and easy to maintain as those without. There is only a one-time overhead of learning the REVCOM protocol for interactions between the iterative method subroutine and the reverse communication interface subroutine.

The overheads resulting from the REVCOM interface are as follows:

1. SAVE statement in the iterative method subroutine. A cost reduction can be achieved by finding a smaller subset of CGREVCOM variables that need to be SAVE'd. However, to some extent this overhead will persist.

2. Cost of RETURNs from CGREVCOM to CG, and CALL to CGREVCOM from CG. The amount of CALL/RETURN overhead depends on the problem size. If the problem size is small, the number of iterations is small (fewer CALL/RETURNs), and therefore this overhead is small. Larger problems entail more iterations and thus greater overhead. However, the ratio of this overhead to execution time is a constant.

3 Use of Reverse Communication

The top-level calling sequence for the iterative method depends on the user's specification of the matrix-vector multiply, preconditioner solve routine(s), and data structures for the various matrices. As such, the user's subroutine implementing the reverse communication interface must be given all the details it needs to perform the operations requested of it. In our example codes, since we are still using COMMON blocks, the top-level calling sequence remains the same. The call to (for example) the CG routine is as follows:

```
      CALL CG( N, B, X(1,1), WORK, LDW, ITER(1), RESID(1),
     $             MATVEC, PSOLVE, INFO(1) )
```

In the reverse communication scheme, the CG subroutine now contains the following call:

```
      CALL CGREVCOM(N, B, X, WORK, LDW, ITER, RESID, INFO,
     $                    NDX1, NDX2, SCLR1, SCLR2, IJOB)
```

[1]In [1] a case is made that a standard calling sequence can be found for the major components of most iterative methods. This would make it possible to write a general iterative package based on user-supplied routines for the matrix operations. However, this is still less general than the REVCOM approach.

The calling sequence to subroutines at this level is fixed and independent of the data structure.

The user must provide implementations of the various matrix-vector operations and stopping tests required by the iterative method. These implementations should adhere to the prescribed calling sequence. If they do not, the reverse communication level (CG in the above example) subroutine will have to be suitably modified. We note that any changes will be restricted to this subroutine.

Note that we have hard-coded the codes for the vectors in supporting stopping tests in the REVCOM interface. The justification is that the iterative method will change very rarely. Further, any changes in its vectors will require commensurate changes in the interface level. The only requirement is that if any changes are made, the encoding *table* in the interface routine must also be updated. This risk can be eliminated by using *include* files (not standard Fortran 77).

The stopping test implementation depends on the user. The only constraint is that after the stopping test has been computed, and before the iterative method is invoked again, the parameter INFO should contain a 1 if the stopping test was successful, and any other integer if the test was not successful. The example code STOPTEST2.f can be used as a template.

In the previous release of the Templates code, the subroutine corresponding to an iterative method (*itmeth*) resided in file *itmeth.f*. For the reverse communication interface, each *itmeth* has two files; *itmeth.f* which implements the reverse communication interface, and *itmethREVCOM.f* which contains the iterative method skeleton. As such, these changes are invisible to the user.

4 Future Work

A number of imporvements can be made in the current implementation, in particular:

- more comprehensive error reporting,
- examples with different storage formats,
- more preconditioners,
- a cookbook explaining how to customize templates for a data format and stopping test, and
- parallelization (selecting suitable data formats; parallel implementation of MATVEC, PSOLVE, SAXPY, NORM, etc.).

We plan to investigate these in future releases.

5 Obtaining the Software

The software implementing the Templates algorithms is available (in Fortran in double precision) from netlib, a public domain software repository. To retrieve the software, send e-mail of the form

```
mail netlib@ornl.gov
```

On the subject line or the body, type

```
send (enter shar filename here) from templates
```

shar filename	contents
dftemplates.shar	Double Precision Fortran 77 routines

The shar file will be returned via an automatic e-mail handler. Save it to a file called (shar filename). Type sh (shar filename). The files described above will be unpacked in the current directory.

The files may also be obtained via anonymous ftp as follows:

```
ftp netlib2.cs.utk.edu

ftp> cd templates
ftp> binary
ftp> get dftemplates.shar
```

In addition, the file may be acquired through the Web at the URL
http://www.netlib.org/templates/index.html.

References

[1] Steven F. Ashby and Mark K. Seager. A proposed standard for iterative linear solvers. Technical Report UCRL-102860, Lawrence Livermore national Laboratory, 1990.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994.

[3] W. D. Joubert, G. F. Carey, N. A. Berner, A. Kalhan, H. Kohli, A. Lorber, R. T. McLay, and Y. Shen. *PCG Reference Manual: A Package for the Iterative Solution of Large Sparse Linear Systems on Parallel Computers*. Report CNA-274, Center for Numerical Analysis, University of Texas, Austin, TX, Group C3, LANL, NM, 1995.

Appendix: Example Code

This appendix contains the REVCOM code for the CG algorithm. The methods use stopping criterion #2. The structure of the matrices is made available to MATVEC, PSOLVE via a COMMON block in the main program.

5.1 CG.f

```
*
      SUBROUTINE CG( N, B, X, WORK, LDW, ITER, RESID, MATVEC,
     $               PSOLVE, INFO )
*
*  -- Iterative template routine --
*     .. Scalar Arguments ..
      INTEGER            N, LDW, ITER, INFO
      DOUBLE PRECISION   RESID
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   X( * ), B( * ),  WORK( * )
*     ..
*     .. Function Arguments ..
      EXTERNAL           MATVEC, PSOLVE
*     ..
*
*  Purpose
*  =======
*
*  CG implements the reverse communication interface.
*  CG invokes CGREVCOM, which returns requests for MATVEC, and
*  PSOLVE. The  appropriate routines are called by CG, and
*  the results are passed to CGREVCOM via another invocation.
*
*  Arguments
*  =========
*  ============================================================
*
*
*   This variable used to communicate requests between CG()
*   and CGREVCOM()
*   CG -> CGREVCOM:  1 = init,
*                    2 = use saved state to resume flow.
*   CGREVCOM -> CG: -1 = done, return to main,
*                    1 = matvec using SCLR1/2, NDX1/2
*                    2 = solve using NDX1/2
*                    3 = matvec using SCLR1/2, NDX1, and vector X
```

```
*                    4 = stop test #2
      INTEGER          IJOB
      LOGICAL          FTFLG
*     Arg/Result indices into WORK[].
      INTEGER          NDX1, NDX2
*     Scalars passed from CGREVCOM to CG.
      DOUBLE PRECISION SCLR1, SCLR2
*     Vars reqd for STOPTEST2
      DOUBLE PRECISION TOL, BNRM2
*     ..
*     .. Executable Statements ..
*
      INFO = 0
*
*     Test the input parameters.
*
      IF ( N.LT.0 ) THEN
         INFO = -1
      ELSE IF ( LDW.LT.MAX( 1, N ) ) THEN
         INFO = -2
      ELSE IF ( ITER.LE.0 ) THEN
         INFO = -3
      ENDIF
      IF ( INFO.NE.0 ) RETURN
*
*     1 == R; 2 == Z; 3 == P; 4 == Q; -1 == ignore; any other == error
*
      NDX1 = 1
      NDX2 = -1
      TOL = RESID
      FTFLG = .TRUE.
*
*     First time call always init.
*
          IJOB = 1

 1        CONTINUE

          CALL CGREVCOM(N, B, X, WORK, LDW, ITER, RESID, INFO,
    $                   NDX1, NDX2, SCLR1, SCLR2, IJOB)

*         On a return from CGREVCOM() we use the table (CGREVCOM -> CG)
*         to figure out what is reqd.
          IF (IJOB .eq. -1) THEN
*           revcom wants to terminate, so do it.
```

```
            GOTO 2
          ELSEIF (IJOB .eq. 1) THEN
*           call matvec.
            CALL MATVEC(SCLR1, WORK(NDX1), SCLR2, WORK(NDX2))
          ELSEIF (IJOB .eq. 2) THEN
*           call solve.
            CALL PSOLVE(WORK(NDX1), WORK(NDX2))
          ELSEIF (IJOB .eq. 3) THEN
*           call matvec with X.
            CALL MATVEC(SCLR1, X, SCLR2, WORK(NDX2))
          ELSEIF (IJOB .EQ. 4) THEN
*           do stopping test 2
*           if first time, set INFO so that BNRM2 is computed.
            IF( FTFLG ) INFO = -1
            CALL STOPTEST2(N, WORK(NDX1), B, BNRM2, RESID, TOL, INFO)
            FTFLG = .FALSE.
          ENDIF
*
*         Done what revcom asked, set IJOB & go back to it.
*
          IJOB = 2
          GOTO 1
*
*     Terminate
*
 2    CONTINUE

      RETURN
*
*     End of CG
*
      END
```

5.2 CGREVCOM.f

```
*
      SUBROUTINE CGREVCOM( N, B, X, WORK, LDW, ITER, RESID, INFO,
     $                     NDX1, NDX2, SCLR1, SCLR2, IJOB)
*
*  -- Iterative template routine --
*     .. Scalar Arguments ..
      INTEGER            N, LDW, ITER, INFO
      DOUBLE PRECISION   RESID
*
*     (output)
```

```
      DOUBLE PRECISION   SCLR1, SCLR2
*     (input/output)
      INTEGER IJOB
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   X( * ), B( * ),  WORK( LDW,* )
*
*     (output) for matvec and solve. These index into WORK[]
      INTEGER NDX1, NDX2
*     ..
*
*  Purpose
*  =======
*
*  CGREVCOM solves the linear system Ax = b using the
*  Conjugate Gradient iterative method with preconditioning.
*  Whenever it needs to compute a MATVEC or PSOLVE, it returns
*  control to CG.
*
*  ============================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ZERO, ONE
      PARAMETER        ( ZERO = 0.0D+0, ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      INTEGER          MAXIT, R, Z, P, Q, NEED1, NEED2
      DOUBLE PRECISION TOL, ALPHA, BETA, RHO, RHO1, BNRM2, DDOT, DNRM2
*
*     Indicates where to resume from. Only valid when IJOB = 2!
      INTEGER RLBL
*
*     Saving all.
      SAVE
*     ..
*     .. External Routines ..
      EXTERNAL         DAXPY, DCOPY, DDOT, DNRM2
*     ..
*     .. Executable Statements ..
*
*     Entry point, test IJOB
      IF (IJOB .eq. 1) THEN
         GOTO 1
      ELSEIF (IJOB .eq. 2) THEN
*        here we do resumption handling
```

```
            IF (RLBL .eq. 2) GOTO 2
            IF (RLBL .eq. 3) GOTO 3
            IF (RLBL .eq. 4) GOTO 4
*           If none of these, error
            INFO = -4
            GOTO 20
         ENDIF
*
*****************
 1       CONTINUE
*****************
*
         INFO = 0
         MAXIT = ITER
         TOL   = RESID
*
*        Alias workspace columns.
*
         R = 1
         Z = 2
         P = 3
         Q = 4
*
*        Check if caller will need indexing info.
*
         IF( NDX1.NE.-1 ) THEN
            IF( NDX1.EQ.1 ) THEN
               NEED1 = ((R - 1) * LDW) + 1
            ELSEIF( NDX1.EQ.2 ) THEN
               NEED1 = ((Z - 1) * LDW) + 1
            ELSEIF( NDX1.EQ.3 ) THEN
               NEED1 = ((P - 1) * LDW) + 1
            ELSEIF( NDX1.EQ.4 ) THEN
               NEED1 = ((Q - 1) * LDW) + 1
            ELSE
*              report error
               INFO = -5
               GO TO 20
            ENDIF
         ELSE
            NEED1 = NDX1
         ENDIF
*
         IF( NDX2.NE.-1 ) THEN
            IF( NDX2.EQ.1 ) THEN
```

```
            NEED2 = ((R - 1) * LDW) + 1
         ELSEIF( NDX2.EQ.2 ) THEN
            NEED2 = ((Z - 1) * LDW) + 1
         ELSEIF( NDX2.EQ.3 ) THEN
            NEED2 = ((P - 1) * LDW) + 1
         ELSEIF( NDX2.EQ.4 ) THEN
            NEED2 = ((Q - 1) * LDW) + 1
         ELSE
*           report error
            INFO = -5
            GO TO 20
         ENDIF
      ELSE
         NEED2 = NDX2
      ENDIF
*
*     Set initial residual.
*
      CALL DCOPY( N, B, 1, WORK(1,R), 1 )
      IF ( DNRM2( N, X, 1 ).NE.ZERO ) THEN

*********CALL MATVEC( -ONE, X, ONE, WORK(1,R) )
*
*        Set args for revcom return
*
         SCLR1 = -ONE
         SCLR2 = ONE
         NDX1 = -1
         NDX2 = ((R - 1) * LDW) + 1
*
*        Prepare for return to CG
*
         RLBL = 2
         IJOB = 3
         RETURN
*
*****************
 2       CONTINUE
*****************
*
         IF ( DNRM2( N, WORK(1,R), 1 ).LT.TOL ) GO TO 30
      ENDIF

      BNRM2 = DNRM2( N, B, 1 )
      IF ( BNRM2.EQ.ZERO ) BNRM2 = ONE
```

```
*
      ITER = 0
*
   10 CONTINUE
*
*        Perform Preconditioned Conjugate Gradient iteration.
*
         ITER = ITER + 1
*
*        Preconditioner Solve.
*
*********CALL PSOLVE( WORK(1,Z), WORK(1,R) )
*
*        Set args for revcom return
*
         NDX1 = ((Z - 1) * LDW) + 1
         NDX2 = ((R - 1) * LDW) + 1
*
*        Prepare for return & return
*
         RLBL = 3
         IJOB = 2
         RETURN
*
*****************
 3       CONTINUE
*****************
*
         RHO = DDOT( N, WORK(1,R), 1, WORK(1,Z), 1 )
*
*        Compute direction vector P.
*
         IF ( ITER.GT.1 ) THEN
            BETA = RHO / RHO1
            CALL DAXPY( N, BETA, WORK(1,P), 1, WORK(1,Z), 1 )
*
            CALL DCOPY( N, WORK(1,Z), 1, WORK(1,P), 1 )
         ELSE
            CALL DCOPY( N, WORK(1,Z), 1, WORK(1,P), 1 )
         ENDIF
*
*        Compute scalar ALPHA (save A*P to Q).
*
*********CALL MATVEC( ONE, WORK(1,P), ZERO, WORK(1,Q) )
*
```

```
*         Set args for revcom return
*
          NDX1 = ((P - 1) * LDW) + 1
          NDX2 = ((Q - 1) * LDW) + 1
          SCLR1 = ONE
          SCLR2 = ZERO
*
*         Prepare for return to CG
*
          RLBL = 4
          IJOB = 1
          RETURN
*
*****************
 4        CONTINUE
*****************
*
          ALPHA =  RHO / DDOT( N, WORK(1,P), 1, WORK(1,Q), 1 )
*
*         Compute current solution vector X.
*
          CALL DAXPY( N, ALPHA, WORK(1,P), 1, X, 1 )
*
*         Compute residual vector R, find norm,
*         then check for tolerance.
*
          CALL DAXPY( N, -ALPHA,  WORK(1,Q), 1, WORK(1,R), 1 )
*
*********RESID = DNRM2( N, WORK(1,R), 1 ) / BNRM2
*********IF ( RESID.LE.TOL ) GO TO 30
*
          NDX1 = NEED1
          NDX2 = NEED2
*         Prepare for resumption & return
          RLBL = 5
          IJOB = 4
          RETURN
*
*****************
 5        CONTINUE
*****************
          IF( INFO.EQ.1 ) GO TO 30
*
          IF ( ITER.EQ.MAXIT ) THEN
             INFO = 1
```

```
            GO TO 20
         ENDIF
*
         RHO1 = RHO
*
         GO TO 10
*
   20 CONTINUE
*
*     Iteration fails.
*
      RLBL = -1
      IJOB = -1
      RETURN
*
   30 CONTINUE
*
*     Iteration successful; return.
*
      RLBL = -1
      IJOB = -1
      RETURN
*
*     End of CGREVCOM
*
      END
```

5.3 STOPTEST2.f

```
*
      SUBROUTINE STOPTEST2( N, R, B, BNRM2, RESID, TOL, INFO )
*
*  -- Iterative template routine --
*     .. Scalar Arguments ..
      INTEGER            N, INFO
      DOUBLE PRECISION   RESID, TOL, BNRM2
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   R( * ), B( * )
*     ..
*
*  Purpose
*  =======
*
*  Computes the stopping criterion 2: ( norm( b - A*x ) / norm( b ) ) < TOL.
```

```
*
* =============================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ZERO, ONE
      PARAMETER        ( ZERO = 0.0D+0, ONE = 1.0D+0 )
*     ..
*     .. External Routines ..
      DOUBLE PRECISION DNRM2
      EXTERNAL         DNRM2
*     ..
*     .. Executable Statements ..
*
      IF( INFO.EQ.-1 ) THEN
         BNRM2 = DNRM2( N, B, 1 )
         IF ( BNRM2.EQ.ZERO ) BNRM2 = ONE
      ENDIF
*
      RESID = DNRM2( N, R, 1 ) / BNRM2
      INFO = 0
      IF ( RESID.LE.TOL ) INFO = 1
*
      RETURN
*
*     End of STOPTEST2
*
      END
```