IDRS package manual

Martin van Gijzen,   
Delft Institute of applied mathematics  
Delft University of technology  
E-mail: M.B.vangijzen@TUdelft.nl  
  
Date: May 2, 2025

# overview of the package

## What is IDR(s)?

IDR(s) is a family of Krylov subspace methods for solving large and sparse nonsymmetric square linear systems of equations

The "IDR" in the names denotes Induced Dimension Reduction. The "S" stand for the iteration parameter s. This parameter determines the amount of vector operations and storage required by the method. A larger s normally increases the rate of convergence, but comes at the price of more memory usage and computations per iteration. For most problems a small value of s (for exampel 2 or 4) gives the best compromise. IDR(s) methods are Krylov subspace methods that use a recursion depth of s+2. This in contrast to most popular Krylov methods for solving nonsymmetric linear systems: BiCGSTAB and GMRES. BiCGSTAB uses three-term recursions while GMRES uses long recursions (recursion depth is equal to the iteration number plus one). IDR(s) methods terminate using at most *N+N*/s iterations, with N the problem size. By comparison, BiCGSTAB terminates within 2*N* iterations and GMRES within *N* iterations. IDR(s) bridges between BiCGSTAB and GMRES, and in fact IDR(s) methods can be obtained that are mathematically equivalent to BiCGSTAB and GMRES.

Next to the standard linear system, specialized versions of IDR(s) have been developed for solving sequences of shifted problems

In which *si*is a scalar shift, and *Ns* is the number of shifts.

## Language

The IDRS package is written in modern Fortran and adheres to the 2018 standard. It is stand-alone, no external library is needed. Coarray Fortran is used for parallelisation. Coarray Fortran is part of the Fortran standard, so the package should run on any platform, from laptop to HPC computer, without modifications. However, some compilers (for example gfortran) require an external library for the programs to run in parallel.

## IDR(s) algorithms included in the package

The package has implementations of the following algorithms:

* IDR(s): an efficient algorithm that uses bi-orthogonality conditions to compute new iteration vectors. Convergence may be erratic. The algorithms is optimised for parallel processing and has only one synchronisation point per iteration. Spectral information can be obtained, and an initial search space can be provided to speed-up the convergence. This is useful when solving sequences of linear systems. With s=1 a method that is mathematically equivalent to BiCGSTAB can be obtained. This IDR(1)-version of BiCGSTAB is exactly as efficient in terms of memory and computations as BiCGSTAB. However, BiCGSTAB might seem faster in terms of the number of iterations. This is only cosmetic, since one BiCGSTAB iteration corresponds to two IDR(1) iterations.
* QMRIDR(s): a robust algorithm that uses orthogonalisation conditions to compute new iteration vectors. QMRIDR(s) uses "quasy" minimisation of the residual norm: a procedure that mimics the optimal minimisation of the residual norm that is used in GMRES. The convergence is much smoother than for IDR(s). As long as the iteration number is smaller than s, QMRIDR(s) is equivalent to GMRES. However, QMRIDR(s) has extra overhead in memory and vector operations. Once the iterations number is larger than s, QMRIDR(s) can loosly be seen as restarted GMRES augmented with a restarting procedure that ensures finite termination. QMRIDR(s) is a flexible method, and can be used as an inner-outer iterative method. In the IDRS package, IDR(s) can be used as inner iterative method for performing preconditioning operations.

For solving shifted problems for multiple shifts, special "multishift" IDR methods are included in the package.These routines can solve a sequence of shifted problems almost at the same cost as solving one linear system.

* Multishift IDR(s): multishift version of the standard IDR(s) algorithm. Does not include extraction of spectral information and re-using subspace information.
* Multishift QMRIDR(s): multishift QMRIDR(s) version. Can use Multishift IDR(s) as inner iterative method.

## preconditioners

IDRS comes with different types of polynomial preconditioners (Chebyshev, Neumann,...), that can be used in combination with diagonal scaling. These preconditioners have been chosen because of the following reasons:

* The user interface becomes quite simple: only a user defined type and a function that performs the matrix-vector multiplication need to be supplied.
* Polynomial preconditioners have the special property that they can be applied to the multishift problem.
* Polynomial preconditioners are well suited for massively parallel computing.

## More general matrix problems

Sequences of systems with changing right-hand side vectors

can be solved with a specialized routine that first extracts spectral information to make an initial search space during the initial solve and then uses this information to speed-up the solution for the remaining systems.

The generalized shifted problems

And

can be solved using one function call. Systems of these forms typically arise while studying wave propagation in mechanical systems. Here ***K*** is the stiffness matrix, ***C*** the damping matrix, and ***M*** is the mass matrix**.** Internally these generalised problems are reduced to the form of a standard shifted system. If the mass matrix is nondiagonal the action of its inverse is approximated by a simple iterative method.

## matrix formats

Matrices can be supplied in user defined formats. Matrices in the standard formats DENSE, Compressed Row Storage (CRS), and Coordinate (COO) can be converted to the matrix type that is used in IDRS by one simple subroutine call. After that the complete functionality of the IDRS package (solvers, preconditioners, parallelisation, graphics) is available.

## Examples

IDRS comes with a wide range of test problems, in DENSE, COO, CRS format, and with examples with user defined formats. Test problems range from academic (for example random and toeplitz matrices) to realistic test problems from applications (acoustics, pagerank, ocean circulation). They include both real and complex examples, and single linear systems, sequences of linear systems and multishift problems.

## Graphics

There is limited support for graphical output. If gnuplot is installed on your computer, it is possible to plot output graphically: convergence curves, estimation of eigenvalues, and for some problems the solution can be plotted on the screen or can be written to gif or jpeg images.

# installing the package

## Downloading and installing

The package can be cloned from the following github repository: https://github.com/mbvangijzen/IDRS-package. The directories with examples contain makefiles. Modify the makefiles to have them call the correct compiler.

The package requires a Fortran compiler that supports the 2018 standard. It has been tested with gfortran and with the intel-compilers ifort and ifx. The latter two compilers have built-in support for co-arrays. The gfortran compiler with the option coarray=single allows you to correctly compile the package but does not enable parallelization. Parallelization can be enabled using an opencoarrays library, which can be downloaded from <http://www.opencoarrays.org>. The package has been tested with the opencoarrays libraries for Linux and for macOS. Graphical output can be obtained using gnuplot, which can be downloaded from <https://gnuplot.sourceforge.net>.

Note that the opencoarrays library and gnuplot extend the functionality of the package, but they are not required. In fact, no external packages, programs, or libraries are required other than a Fortran compiler that supports the 2018 standard.

## directory structure

Downloading the package on your computer will create the following directory structure:

**IDRS** *root directory*

**MODULES** *The source code that forms the algorithmic heart of the package.*

**IDRS** *Contains the source code of the IDR(s) algorithms.* **INTERFACE** *Contains a general interface module to pass command line parameters* **PP\_IDRS** *Contains a module with wrapper routines that combine preconditioners and solvers* **PP\_MATRIX** Contains modules for the preconditioners and matrix-vector products

**EXAMPLES** *The set of examples to test the algorithms.*

**MATRIX\_MARKET** *test problems in matrix-market format*

**DATA** *needed by some test problems*

**OCEAN\_DATA** *for ocean problems Sag and Stommel in matrix-market format***WEDGE\_DATA** *for Helmholtz problem Wedge*

**PAGERANK** *PageRank test problems in CRS format***RANDOM** *dense random test problems***STENCIL** *test problems**that use stencil-based matrix-vector multiplication* **TOEPLITZ** *test problems with Toeplitz matrices***TOP** *finite elements topology optimization test problems*

# USIng the package, combining it with your own software

## Setting the precision

The precision with which the package calculates is determined by the parameters rp for real numbers, and cp for complex numbers. These parameters set the kind of real and complex numbers, and they should match the precision of the user routines that calls the routines in the package. The parameters rp and cp are set in the module PRECISION\_MODULE, which is in the directory **IDRS**. By default, both the kind parameters are set to real64 (double precision).

## using the package

The package can be used in different ways depending on the functionality that is wanted:

1. Only the basic IDR(s) algorithms will be used. The matrix-vector multiplication and preconditioning operation need to be supplied by the user. A user defined matrix type also needs to be provided.
2. Both the IDR(s) algorithms and the polynomial preconditioners will be used. Only the matrix-vector multiplication needs to be supplied by the users. A user defined matrix type also needs to be provided.
3. Both the IDR(s) algorithms and the polynomial preconditioners will be used, and the matrix is defined in one of the following standard formats: DENSE, COO, CRS. In this case the user only needs use the appropriate routine in the package to convert the matrix data in standard format to a matrix of TYPE(USER\_MATRIX).

### Only the basic IDRs algorithms will be used

If only the basic IDRS algorithms are used, only the modules contained in the directory **IDRS/MODULES/IDRS** are needed.

#### What needs to be supplied by the user?

The user must supply a module MATRIX\_MODULE that defines the matrix-vector multiplication and the preconditioning operation. It should have the following structure:

MODULE MATRIX\_MODULE  
 USE PRECISION MODULE  
 IMPLICIT NONE

TYPE:: MATRIX

! Put here user defined variables needed to perform the matrix-vector   
! and preconditioning operation.

END TYPE MATRIX

! Overload \* to define the matrix-vector multiplication using the matrix type  
 INTERFACE OPERATOR(\*)  
 MODULE PROCEDURE RMATVEC, CMATVEC  
 END INTERFACE

! Overload / to define the preconditioning operation using the matrix type  
 INTERFACE OPERATOR(/)  
 MODULE PROCEDURE RPRECON, CPRECON  
 END INTERFACE

CONTAINS

FUNCTION CMATVEC( A, V ) RESULT(W)  
 TYPE(MATRIX), INTENT(IN) :: A  
 COMPLEX(KIND=CP), INTENT(IN) :: V(:)  
 COMPLEX(KIND=CP) :: W(SIZE(V))  
  
! Add declarations and statements for the complex matrix-vector  
! multiplication

END FUNCTION CMATVEC

FUNCTION RMATVEC( A, V ) RESULT(W)  
 TYPE(MATRIX), INTENT(IN) :: A  
 REAL(KIND=RP), INTENT(IN) :: V(:)  
 REAL(KIND=RP) :: W(SIZE(V))  
  
! Add declarations and statements for the real matrix-vector multiplication

END FUNCTION RMATVEC

FUNCTION CPRECON( A, V ) RESULT(W)  
 TYPE(MATRIX), INTENT(IN) :: A  
 COMPLEX(KIND=CP), INTENT(IN) :: V(:)  
 COMPLEX(KIND=CP) :: W(SIZE(V))  
  
! Add declarations and statements for the complex preconditioning operation

END FUNCTION CPRECON

FUNCTION RPRECON( A, V ) RESULT(W)  
 TYPE(MATRIX), INTENT(IN) :: A  
 REAL(KIND=RP), INTENT(IN) :: V(:)  
 REAL(KIND=RP) :: W(SIZE(V))

! Add declarations and statements for the real preconditioning operation

END FUNCTION RPRECON

END MODULE MATRIX\_MODULE

#### CallING The IDR(S) algorithms

With the MATRIX\_MODULE available, the routines in the IDRS\_MODULE can be used. The module IDRS\_MODULE is at the heart of the package. It contains the four IDRS-algorithms: IDRS, QMRIDR, MSIDRS and MSQMRIDR. IDRS and QMRIDR solve the equation

With **A** the system matrix**, b** the known right-hand side vector, **x** the solution and ***P*** the preconditioner. The system is iteratively solved until the norm of the residual after *k* iterations satisfies

*.*

The iterative process is stopped if the iteration number *k* exceeds the (user specified) maximum number of iterations.

MSIDRS and MSQMRIDR solve the sequence of shifted equations

for a number of shifts si. Here **I** is the identity matrix, and **P** should be viewed as the shift matrix, rather than the preconditioner. These systems are iteratively solved until the norms of the residuals satisfy

*.*

or until *k* exceeds the maximum number of terations that is allowed.

To make the four IDR(s) functions available, add the statement use idrs\_module to the program unit that calls the functions. The functions are called as follows:

Function IDRS

x = IDRS( A, b, s, M1, tolerance, maximum\_iterations, variant, flag, relres, iterations, x0, U0, omega, resvec, H )

The result vector x is a real(kind=rp) or complex(kind=cp) assumed shape array of dimension n, with n the problem size. The parameters are as follows:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Meaning** |
| A | type(matrix), required, input | System matrix |
| b | real(kind=rp) or complex(kind=cp), required, assumed shape array of size n | Right-hand-side vector |
| s | integer > 0, required, input | Size of shadow space |
| M1 | type(matrix), optional, input | Preconditioner |
| tolerance | real(kind=rp), optional, input, default 1e-6 | Termination criterion |
| maximum\_iterations | integer, optional, input, default min(2\*n,1000) | Maximum number of iterations |
| variant | integer, optional, input, default 1 | Variant = 1: idrs, 2: bicgstab |
| flag | integer, optional, output | Convergence flag:  flag = 0: normal termination  flag = 1: maximum iterations  reached  flag = 2: accuracy above   tolerance  flag = 3: break down |
| relres | real(kind=rp), optional, output | Relative residual norm |
| iterations | integer, optional, output | Number of iterations to converge |
| x0 | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n | Initial guess |
| U0 | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n\*s | Initial search space. See also the function ritz\_idrs. |
| omega | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n\_omega | User specified values for omega. See also the function ritz\_idrs |
| resvec | real(kind=rp), assumed shape array of size maximum\_iterations+1, optional, output | Residual norm for every iteration |
| H | real(kind=rp) or complex(kind=cp), optional, output, assumed shape array of size (nritz+1)\*nritz, with nritz the number of wanted ritz values. | Hessenberg matrix whose eigenvalues are the ritz values, see als the function ritz\_idrs |

Table 1. Parameters of function IDRS.

Function QMRIDR

x = QMRIDR( A, b, s, M1, tolerance, maximum\_iterations, flag, relres, iterations, x0, omega, resvec )

The result vector x is a real(kind=rp) or complex(kind=cp) assumed shape array of size n, with n the problem size. The parameters are as follows:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Meaning** |
| A | type(matrix), required, input | System matrix |
| b | real(kind=rp) or complex(kind=cp), required, assumed shape array of size n | Right-hand-side vector |
| s | integer > 0, required, input | Size of shadow space |
| M1 | type(matrix), optional, input | Preconditioner |
| tolerance | real(kind=rp), optional, input, default 1e-6 | Termination criterion |
| maximum\_iterations | integer, optional, input, default min(2\*n,1000) | Maximum number of iterations |
| flag | integer, optional, output | Convergence flag:  flag = 0: normal termination  flag = 1: maximum iterations  reached  flag = 2: accuracy above   tolerance  flag = 3: break down |
| relres | real(kind=rp), optional, output | Relative residual norm |
| iterations | integer, optional, output | Number of iterations to converge |
| inner\_s | integer,optional, input, default 4 | Parameter s for inner iterations |
| inner\_tolerance | real(kind-rp),optional, input, default = 0.1 | Tolerance for inner iterations |
| inner\_iterations | integer, optional, input, default 0 | Maximum number of inner iterations |
| x0 | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n | Initial guess |
| omega | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n\_omega | User defined values for omega. See also the function ritz\_idrs |
| resvec | real(kind=rp), assumed shape array of size maximum\_iterations+1, optional, output | Residual norm for every iteration |

Table 2. Parameters of function QMRIDR.

Function MSIDRS

x = MSIDRS( A, b, sigma, s, M1, tolerance, maximum\_iterations, variant, flag, relres, iterations, omega, resvec, colfac )

The result vector x is a real(kind=rp) or complex(kind=cp) assumed shape array of dimension n\*n\_sigma, with n the problem size and n\_sigma the number of shifts. The parameters are as follows:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Meaning** |
| A | type(matrix), required, input | System matrix |
| b | real(kind=rp) or complex(kind=cp), required, assumed shape array of dimension n | Right-hand-side vector |
| sigma | real(kind=rp) or complex(kind=cp), required, assumed shape array of size n\_sigma | The shifts |
| s | integer > 0, required, input | Size of shadow space |
| M1 | type(matrix), optional, input | Mass matrix |
| tolerance | real(kind=rp), optional, input, default 1e-6 | Termination criterion |
| maximum\_iterations | integer, optional, input, default min(2\*n,1000) | Maximum number of iterations |
| variant | integer, optional, input, default 1 | Variant = 1: msidrs, 2: msbicgstab |
| flag | integer, optional, output | Convergence flag:  flag = 0: normal termination  flag = 1: maximum iterations  reached  flag = 2: accuracy above   tolerance  flag = 3: break down |
| relres | real(kind=rp), assumed shape array of size n\_sigma , optional, output | Relative residual norms |
| iterations | integer, optional, output | Number of iterations to converge |
| omega | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n\_omega | User specified values for omega. See also the function ritz\_idrs. |
| resvec | real(kind=rp), assumed shape array of size (maximum\_iterations+1)\*n\_sigma, optional, output | Residual norms for every iteration |
| colfac | real(kind=rp) or complex(kind=cp), optional, output, assumed shape array of size n\_sigma | Collinearity factors, only for internal use |

Table 3. Parameters of function MSIDRS.

Function MSQMRIDR

x = MSQMRIDR( A, b, sigma, s, M1, tolerance, maximum\_iterations, flag,relres, iterations, inner\_s, inner\_tolerance, inner\_iterations, omega, resvec )

The result vector x is a real(kind=rp) or complex(kind=cp) assumed shape array of dimension n\*n\_sigma, with n the problem size and n\_sigma the number of shifts. The parameters are as follows:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Meaning** |
| A | type(matrix), required, input | System matrix |
| b | real(kind=rp) or complex(kind=cp), required, assumed shape array of dimension n | Right-hand-side vector |
| sigma | real(kind=rp) or complex(kind=cp), required, assumed shape array of size n\_sigma | The shifts |
| s | integer > 0, required, input | Size of shadow space |
| M1 | type(matrix), optional, input | Mass matrix |
| tolerance | real(kind=rp), optional, input, default 1e-6 | Termination criterion |
| maximum\_iterations | integer, optional, input, default min(2\*n,1000) | Maximum number of iterations |
| variant | integer, optional, input, default 1 | Variant = 1: msidrs, 2: msbicgstab |
| flag | integer, optional, output | Convergence flag:  flag = 0: normal termination  flag = 1: maximum iterations  reached  flag = 2: accuracy above   tolerance  flag = 3: break down |
| relres | real(kind=rp), assumed shape array of size n\_sigma , optional, output | Relative residual norms |
| iterations | integer, optional, output | Number of iterations to converge |
| inner\_s | integer,optional, input, default 4 | Parameter s for inner iterations |
| inner\_tolerance | real(kind-rp),optional, input, default = 0.1 | Tolerance for inner iterations |
| inner\_iterations | integer, optional, input, default 0 | Maximum number of inner iterations |
| omega | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n\_omega | User specified values for omega. See also the function ritz\_idrs. |
| resvec | real(kind=rp), assumed shape array of size (maximum\_iterations+1)\*n\_sigma, optional, output | Residual norms for every iteration |

Table 4. Parameters of function MSQMRIDR.

#### RITZ\_idrs

The four IDR(s) algorithms described in the previous section have as optional input user supplied parameters omega, and (only for IDRS) an initial search space U0. The function RITZ\_IDRS can be used to compute suitable values for these parameters. RITZ\_IDRS itself calls IDRS to perform several IDRS-iterations and obtain the Hessenberg matrix H. The eigenvalues of this matrix are called Ritz values and are approximations to the eigenvalues of the preconditioned system matrix. By using the information contained in H, RITZ\_IDRS computes suitable omega’s and an initial search space U0. It also returns the approximate solution after the initial iterations, and the matrix H itself.

To make RITZ\_IDRS available, add the statement use ritz\_module to the program unit that calls the function. RITZ\_IDRS is called as follows:

Function RITZ\_IDRS

x = RITZ\_IDRS( A, b, s, M1, tolerance, maximum\_iterations, variant, flag, relres, iterations, x0, U0, omega, resvec, H )

The result vector x is a real(kind=rp) or complex(kind=cp) assumed shape array of dimension n, with n the problem size. The parameters are as follows:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Meaning** |
| A | type(matrix), required, input | System matrix |
| b | real(kind=rp) or complex(kind=cp), required, assumed shape array of size n | Right-hand-side vector |
| s | integer > 0, required, input | Size of shadow space |
| M1 | type(matrix), optional, input | Preconditioner |
| tolerance | real(kind=rp), optional, input, default 1e-6 | Termination criterion |
| maximum\_iterations | integer, optional, input, default min(2\*n,1000) | Maximum number of iterations |
| variant | integer, optional, input, default 1 | Variant = 1: idrs, 2: bicgstab |
| flag | integer, optional, output | Convergence flag:  flag = 0: normal termination  flag = 1: maximum iterations  reached  flag = 2: accuracy above   tolerance  flag = 3: break down |
| relres | real(kind=rp), optional, output | Relative residual norm |
| iterations | integer, optional, output | Number of iterations to converge |
| x0 | real(kind=rp) or complex(kind=cp), optional, input, assumed shape array of size n | Initial guess |
| U0 | real(kind=rp) or complex(kind=cp), optional, output, assumed shape array of size n\*s | Initial search space, based on Ritz vectors. |
| omega | real(kind=rp) or complex(kind=cp), optional, output, assumed shape array of size n\_omega | Suggested values for omega base on Ritz values |
| resvec | real(kind=rp), assumed shape array of size maximum\_iterations+1, optional, output | Residual norm for every iteration |
| H | real(kind=rp) or complex(kind=cp), optional, output, assumed shape array of size (nritz+1)\*nritz, with nritz the number of wanted ritz values. | Hessenberg matrix |

Table 5. Parameters of function RITZ\_IDRS.

### Both the IDR(s) algorithms and the polynomial preconditioners will be used.

If also the polynomial preconditioners are used, the module MATRIX\_MODULE contained in the directory **PP\_MATRIX** is needed. The IDR(s) algorithm can be combined with the polynomial preconditioners in a very simple way using the wrapper routines contained in the module PP\_IDRS, which is contained in the directory **PP\_IDRS**.

#### The module MATRIX\_MODULE

The module MATRIX\_MODULE (implicitly) constructs the system matrix ***A*** from a matrix triplet (***K***,***C***, ***M***), and a scalar shift τ. Here ***K*** is the stiffness matrix, ***C*** the damping matrix, and ***M*** the mass matrix. ***C****,* ***M*** and *τ* are optional, by default they are zero.

Standard problem

The standard problem has the general form

The specific systems that can be solved, depending which parameters are specified, are:

**,**

and

Multishift problem

The multishift problem has the general form

The specific systems that can be solved, depending which parameters are specified, are:

,

,

and

.

The first problem is in the appropriate form of a multishift problem, i.e. the shift matrix is equal to the identiy. The second and third problem, however, are not and first need to be reformulated into equivalent multishift problems. This is done as follows

,

and

The corresponding matrices ***A*** are

and

respectively. Note that

Multishift problem

Preconditioners

A polynomial preconditioner in combination with a diagonal scaling matrix **D** can be used to speed-up the convergence.This preconditionerhas the form

**P** = **D**-1*Pd*(**A**(**K**, **C**, **M,** τ)**D**-1).

Here *Pd*(.) can be either a Chebyshev or a Neumann polymial. For the standard problem, a suitable choice for **D** is

**D** = diag(**A**).

#### What needs to be supplied by the user?

The user must supply a module USER\_MODULE that defines the matrix type USER\_MATRIX for the matrices ***K****,* ***C****,* and ***M*,** and the matrix-vector for matrices of this type. It should have the following structure:

MODULE USER\_MODULE  
 USE PRECISION MODULE  
 IMPLICIT NONE

TYPE:: USER\_MATRIX

! Put here user defined variables needed to perform the matrix-vector   
! and preconditioning operation.

END TYPE USER\_MATRIX

! Overload \* to define the matrix-vector multiplication using the matrix type  
 INTERFACE OPERATOR(\*)  
 MODULE PROCEDURE RMATVEC, CMATVEC  
 END INTERFACE

CONTAINS

FUNCTION CUSER\_MV( A, V ) RESULT(W)  
 TYPE(MATRIX), INTENT(IN) :: A  
 COMPLEX(KIND=CP), INTENT(IN) :: V(:)  
 COMPLEX(KIND=CP) :: W(SIZE(V))  
  
! Add declarations and statements for the complex matrix-vector  
! multiplication

END FUNCTION CUSER\_MV

FUNCTION RUSER\_MV( A, V ) RESULT(W)  
 TYPE(MATRIX), INTENT(IN) :: A  
 REAL(KIND=RP), INTENT(IN) :: V(:)  
 REAL(KIND=RP) :: W(SIZE(V))  
  
! Add declarations and statements for the real matrix-vector multiplication

END FUNCTION RUSER\_MV

END MODULE USER\_MODULE