University of Maryland College Park

Institute for Advanced Computer Studies TR–2003–89
Department of Computer Science TR–4521

MATRAN
A Fortran 95 Matrix Wrapper*

G. W. Stewart†

August 2003

ABSTRACT

MATRAN is an wrapper written in Fortran 95 that implements matrix operations and computes matrix decompositions using LAPACK and the BLAS. This document describes a preliminary release of MATRAN, which treats only real matrices. Its purpose is to get outside comments and suggestions before the package jells. Consequently, this documentation is slanted toward the experienced programmer familiar with both matrix computations and Fortran 90/95. User oriented documentation will accompany the final release.

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

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (`stewart@cs.umd.edu`). This work was supported in part by the National Science Foundation under grant CCR0204084.

Contents

Preface

This document introduces a preliminary version of MATRAN (pronounced MAY-tran), a Fortran 95 wrpper that implements matrix operations and computes matrix decompositions using LAPACK and the BLAS. Although MATRAN is not based on a formally defined matrix language, it provides the flavor and convenience of coding in matrix oriented systems like MATLAB, OCTAVE, etc. By using routines from LAPACK and the BLAS, MATRAN allows the user to obtain the computational benefits of these packages with minimal fuss and bother.

MATRAN originated as follows. In 2002, my colleague Dianne O'Leary and I received an NSF grant to work on new algorithms for large-scale eigenvalue problems. Somewhat rashly we promised to implement our algorithms in a standard high level language, even though we knew that we would develop them using MATLAB. A couple of years previously I had published a Java matrix package called JAMPACK. The response was less than enthusiastic, owing in part to the awkward syntax forced on it by the absence of operator overloading in Java. Since Fortran 95 not only can overload operators but can also can define new ones, it occurred to me that JAMPACK would look a lot cleaner in Fortran 95 and could, in fact, provide natural and efficient implementations of code from matrix oriented languages.

At present MATRAN implements only real matrix operations and decompositions. Consequently, it is still is small enough to survive significant changes, provided they represent substantial improvements. The purpose of this release is to solicit comments and suggestions before MATRAN jells. For this reason, this document is addressed largely to experts — people well grounded in matrix computations, Fortran 95, LAPACK, and the BLAS. The formal release, which will contain complex types, will be accompanied by a more conventional user's manual.

MATRAN may be obtained through my home page

`http://www.cs.umd.edu/~stewart/`

This project has many benefactors. I am supported by the National Science Foundation at the Computer Science Department and the Institute for Advanced Computer Studies of the University of Maryland. I am also a faculty appointee at the Mathematical and Computational Sciences Division of the National Institute for Standards and Technology, where my division leader, Ron Boviert, has encouraged me to work on this project.

I am greatly indebted to John Reid, who patiently steered me through my initial fumblings with Fortran 95 and provided useful suggestions for the design of MATRAN. His excellent book with Michael Metcalf, *Fortran 90/95 Explained,* has been my constant companion during this project. Bill Mitchel, the resident NIST expert on Fortran 90/95, has made himself cheerfully available on a drop-in basis to answer my questions. Finally, my student Che-Rung Lee, who came in at the middle of the project and quickly learned the ropes, has been a valuable assistant ever since.

MATRAN
A Fortran 95 Wrapper for Matrix Operations

G. W. Stewart

1. Overview and example

MATRAN is an open wrapper written in Fortran 95 that implements matrix operations and computes matrix decompositions using LAPACK and the BLAS. MATRAN is a blending of "matrix" and "Fortran," and is pronounced MAY-tran. This document describes a preliminary release of MATRAN which treats only real matrices. Its purpose to get outside comments and suggestions before the package jells. Consequently, it is slanted toward the experienced programmer familiar with both matrix computations and Fortran 90/95. User oriented documentation will accompany the final release.

1.1. Overview

MATRAN is a collection of derived types and generic subprograms in Fortran 95 that implements matrix operations and computes matrix functions and decompositions. Although MATRAN is not based on a formally defined matrix language, the results of using MATRAN are akin to coding in a subset of matrix oriented programming languages like MATLAB, OCTAVE, etc. By using routines from LAPACK and the BLAS, MATRAN allows the user to obtain the computational benefits of these packages with minimal fuss and bother.

Here are some of the features of MATRAN.

- This preliminary release of MATRAN provides only two matrix types. The `Rmat` represents matrices stored in rectangular arrays. The `Rdiag` implements diagonal matrices stored in a linear array.[1] However, this poverty of types is illusory. The type `Rmat` contains a tag field that subdivides the type into general, upper triangular, lower triangular, symmetric, and symmetric positive definite matrices. The first formal release will also include complex versions of the two types. Ultimately, I would like to see MATRAN support band and sparse matrices.

- There are single and double versions of MATRAN, corresponding to the single and double precision versions of LAPACK and the BLAS. The default result of compilation is double precision; but compilation of a single precision package can be forced by setting a flag in the compilation command line. Unfortunately, one cannot mix or match: the

[1] In Fortran 95 these arrays are said to have *rank* two and one respectively. However, since the word rank has other meanings in matrix computations, we use the terms rectangular and linear instead.

package is all single precision or all double precision. Incidently, if LAPACK quad codes become avaliable, it will be easy to extend MATRAN to a quad package.

• Matrix operations are provided by overloaded and defined operators. For example `A + B` compute the sum of the matrices A and B, while `A.xhy.B` computes $A^{\mathrm{H}}B$. A suite of subprograms computes products like $A^{-1}B$ or $A^{-\mathrm{H}}B$. In addition, MATRAN defines operations for combining matrices and extracting submatrices.

• MATRAN provides common matrix functions — e.g., norms — as well as constructors for special matrices like the identity.

• Matrix types in MATRAN are allowed to be void (aka empty) — that is, they may have zero row or column dimension (or both). This feature is useful in starting matrix algorithms that build up matrices by bordering.

• MATRAN provides types for the following decompositions: the pivoted LU decomposition, the Cholesky decomposition, the pivoted and unpivoted QR decompositions, the spectral decomposition of a symmetric matrix, the singular value decomposition, and the Schur and eigendecompositions of a general square matrix. MATRAN provides means for reusing decompositions, as, for example, when one wishes to solve several linear systems all having the same matrix.

• MATRAN is modularized at a fine-grained level. This means that the programmer can pick and choose among MATRAN's capabilities without linking to the entire package.

• Storage management in MATRAN requires only a minimal assist from the user. However, MATRAN provides additional means by which the user can force the reuse of storage already allocated, thus reducing calls to the allocator. These features may be useful to people doing large computations with small matrices, in which the allocation of intermediate matrices can amount to a significant part of the computational load.

• Many of MATRAN's more advanced features are implemented via optional arguments, so that when they are not needed they do not clutter the code.

• MATRAN is an *open* package in the sense that its modules and types have no private components. This fact has two useful consequences.

1. The programmer can use the resources of Fortran 95 to manipulate matrices in ways not provided by MATRAN. This ability is especially important for matrix computations, since the number of things people want to do with matrices far exceeds the number of methods that a closed, object-oriented package can provide.

2. Closely related to the first is the fact that the programmer can do things in a way that facilitates compiler optimization. To give a single example, a `Rmat` holds its matrix in an array called `a`. In MATRAN, the standard way to reference the

> `(i,j)`-element of a `Rmat M` is `M%a(i,j)`, which means the the compiler knows that it is working with references to a rectangular array and can optimize the code accordingly. If access were exclusively through functions, the compiler would not be able to optimize.

However, there is a downside to being open. MATRAN cannot enforce its own conventions. Thus the MATRAN programmer must be more both knowledgeable and more disciplined than the casual user of object-oriented packages.

1.2. A least squares solver

In this section we will illustrate some of MATRAN's features and conventions by a simple least squares solver. Suppose we are given an $m \times n$ matrix A of full column rank n. Given an m-vector b we want to compute an n-vector x such that

$$\|b - Ax\|_2^2 = \min,$$

where $\|u\|_2^2 = \sum_i u_i^2$. In addition, we want to compute the residual $r = b - Ax$ at the minimum, and the residual sum of squares $\|r\|_2^2$.

The QR decomposition furnishes an elegant way of solving this problem. Specifically, we can write A in the form

$$A = QR, \tag{1.1}$$

where Q has orthonormal columns and R is upper triangular. It can be shown that

$$x = R^{-1}Q^{\mathrm{T}}b.$$

Hence, given the QR decomposition of A, one can find x by simple operations involving b, Q, and R.

The code in Figure 1.1 implements this algorithm. The statement

```
use MatranRealCore_m
```

invokes a blanket module consisting of `use` statements invoking the core modules of MATRAN (§7).[2] The second `use` statement gets the module defining the QR decomposition and its constructor.

The variables A, b, x, and r have changed to the `Rmat`s `A`, `b`, `x`, and `r`. A `Rmat` is a defined type that implements a matrix as a set of numbers stored in a rectangular array in the usual way. We will have more to say about `Rmat`s later. But note that MATRAN makes no distinction between matrices and vectors. The are all represented by the same derived type — the `Rmat`.

[2] In MATRAN all modules are suffixed with `_m`.

```
subroutine qrlsq(A, b, x, r, RSS)

use MatranRealCore_m
use RmatQR_m
implicit none

   type(Rmat), intent(in) :: A, b
   type(Rmat), intent(out) :: x, r
   real(wp), intent(out) :: RSS

   type(RmatQR) :: qra

   !Protect temporaries.

   call GuardTemp(A); call GuardTemp(b)

   ! Get the QR decomposition of A.

   call QR(qra, A)

   ! Solve the least squares problem.

   x = qra%R.xiy.(qra%Q.xhy.b)
   r = b - A*x
   RSS = NormF(r)**2

   ! Clean up.

   call Clean(qra)
   call CleanTemp(A); call CleanTemp(b)

end subroutine qrlsq
```

Figure 1.1: QR least squares

The residual sum of squares is returned via the paramenter `RSS`. It is declared to be a real scalar of kind `wp`. The parameter `wp` (for Working Precision) is defined at compile time in the module MatranUtil_m.

Let begin with the computational heart of the algorithm. The statement

```
call QR(qra, A)
```

computes the QR decomposition of A. In MATRAN this decomposition has the form

```
type RmatQR
   type(Rmat) :: Q
   type(Rmat) :: R
   logical :: companion
end type RmatQR
```

The first two components are Rmats containing the Q- and R-factors of A [cf. (1.1)]. The third component will be discussed later (§6.1).

The computation in the statement

```
x = qra%R.xiy.(qra%Q.xhy.b)
```

consists of two parts. The first part, qra%Q.xhy.b computes $t = Q^{\mathrm{T}}b$. The operator .xhy. is to be read, "x conjugate transpose y," and it means just what it says: the conjugate transpose of the first operand multiplies the second operand. This, of course, is the same as multiplying by the transpose. But MATRAN prefers to specify the conjugate transpose for both real and complex matrices to aid in generalizing programs from real to complex arithmetic. (The practice is similar to the use of the superscript '*' to denote the adjoint of a matrix or operator, whatever the underlying field.)

The second part computes $R^{-1}t$. The operation .xiy. reads "x inverse y." But the "inverse" is there only for brevity, and in fact it is never computed. Instead MATRAN solves the system $Rx = t$. MATRAN is smart enough to recognize that R is upper triangular and use the appropriate algorithm.

The computation of

```
r = b - A*x
```

uses the overloaded operators - and * and is straightforward. However, you can get unexpected results if you combine defined operators with overloaded operators because the latter bind more tightly than the former. For example, the expression a + B.xhy.c computes $(a + B)^{\mathrm{T}}c$, not $a + B^{\mathrm{T}}c$ as expected. To get the latter you must write a + (B.xhy.c). In MATRAN the watchword is: When in doubt, parenthesize.[3]

[3]There is another reason for being careful with parentheses. Suppose A B and C are respectively n x 1, 1 x n and n x n Rmats, and we wish to compute A*B*C. For defined or overloaded operators, Fortran 95 evaluates left to right — i.e., (A*B)*C, an expression which requires $O(n^3)$ floating-point operations to compute. On the other hand, the expression A*(B*C) requires only $O(n^2)$ operations. Thus, in this case, the expression A*B*C should be parenthesized in the form A*(B*C).

Another source of confusion arises from the fact that Fortran makes no distinction between upper and lower case letters. Thus we could have just as well written

```
R = B - a*X
```

This can easily lead to programming errors in matrix computations, where capital letters frequently denote matrices and small letters denote vectors. For example, consider writing code based on a paper in which u represent a column of a matrix U.

Finally, the residual sum of squares is computed as the square of the Frobenius norm of `r`. The function `NormF` is one of a suite of generic fuctions that compute matrix norms.

MATRAN automatically takes care of finding storage to hold the results of its computations. Unfortunately, the user must help with deallocation. This is because MATRAN uses pointer arrays, which are not deallocated automatically, to hold its matrices.[4] The rules for deallocation this are simple. The first rule is

Before returning from a subprogram use the `Clean` *subroutine to deallocate the storage of all locally defined matrix objects and decompositions.* (1.2)

For example, the statement

```
call Clean(qra)
```

in our sample program deallocates storage for the `Rmat`s `qra%Q` and `qra%R`.

The second rule addresses a more subtle problem. Consider once again the statement

```
r = b - A*x
```

The first thing that must be computed is the quantity `A*x`, which in MATRAN is a `Rmat`. This temporary `Rmat` — call it `t` — is no longer needed after it is used to compute `b - t`, and MATRAN silently deallocates it. Likewise another temporary `Rmat` is needed to hold `b - t` before it is copied to `r`. Once again, MATRAN silently allocates and deallocates the temporary.

The problem comes when you invoke a subprogram with a temporary for an actual argument. For example, one might call `qrlsq` as follows.

```
call qrlsq(A, c-d, x, r)
```

[4]The reason is that strict Fortran 95 does not allow allocatable arrays appear in defined types. There is an extension of Fortran 95, guaranteed to be in the Fortran 200x standard, that allows such constructions; but it is not everywhere implemented. In the future MATRAN will use allocatable arrays, and the extension will be backward compatible with code written in accordance with the conventions of the present version.

In this case `c-d` will be a temporary `Rmat` — but one that has cut free from MATRAN, which therefore cannot deallocate it. The cure is contained in the following rule.

> *Just after entering a subprogram call* `GuardTemp` *with each dummy matrix object of the subprogram having the intent* `in`. *Just before leaving, call* `CleanTemp` *with each of the same dummy arguments.* (1.3)

Thus in `qrlsq` we have the statements

```
call GuardTemp(A); call GuardTemp(b)
```

at the beginning and the statements

```
call CleanTemp(A); call CleanTemp(b)
```

at the end.

MATRAN routines are not the only ones that generate temporary variables. Whenever a user defined function returns a MATRAN matrix type, the returned value must be regarded as temporary, since it can only occur in an expression or as an actual parameter in an argument list. The subroutine `SetTemp` declares a matrix to be a temporary.

> *If a function returns a matrix object* `M`, *then execute*
>
> `call SetTemp(M)` (1.4)
>
> *before returning.*

Although these rules may seem involved, they generate very little code. Moreover, the calls to `GuardTemp` occur only at the beginning of the routine in question. If the routine is coded to have only one point of return (presumably at the end), the calls to `ClearTemp` and `SetTemp` occur only at that point.

Finally, as we have noted above, MATRAN uses pointer arrays to store matrices. Eventually, when the Fortran world is sufficiently settled, the pointer arrays will be replaced by allocatable arrays, which will obviate the need for the convention (1.2)–(1.4). However, to be consistent with the change to allocatable arrays, you should not do things with the pointer array of a matrix object that cannot be done with allocatable arrays. In particular, you should observe the following strictures.

> *Neither change the association of nor assign a pointer to the array in a matrix object.* (1.5)

You may, however, allocate and deallocate the pointer arrays of a matrix object. Just make sure you know what you are doing.

Owing to bug in Sun WorkShop 6 update 2 Fortran 95 6.2 2001/05/15, additional initialization has to be done on the result of a function. See §9

2. The module MatranUtil_m

The module `MatranUtil_m` is the root MATRAN module. It contains a parameter for defining the precision of real types, error handlers, and procedures for reshaping raw arrays.

`MatranUtil` defines the parameter `wp` by

```
#ifdef sngl
   integer, parameter :: wp = kind(1.0e0)
#endif
#ifdef dbl
   integer, parameter :: wp = kind(1.0d0)
#endif
```

Thus the specification

```
real(wp) :: <variable list>
```

declares the variables in the list to be of the precision selected for this version of Matran. The default is double precision. The selection is done by defining one of the Fortran preprocessor parameters `sngl` or `dbl`, which can be done at compilation time in the command line. (Actually, if you do nothing, you get double precision.)

The general error handler for MATRAN is

```
subroutine MatranError(ErrorMessage)
```

where

```
character(*), intent(in) ::  ErrorMessage
```

The subroutine prints the error message and stops.

As we have mentioned, MATRAN uses LAPACK and the BLAS to perform most of its calculations. The former returns error indications via a standard parameter `info`. In case of such an error, MATRAN uses the following error handler.

```
\begin{frag}
subroutine SupportError(ErrorMessage, infonum)
\end{frag}
```

where

```
character(*), intent(in) ::  ErrorMessage
integer, intent(in) ::  infonum
```

The subroutine prints the error message followed by

```
subroutine ReshapeAryD2(Ary, m, n)
   real(wp), pointer   :: Ary(:,:)
   integer, intent(in) :: m, n

   integer :: shp(2)

   if (associated(Ary)) then
      shp = shape(Ary)
      if (m>shp(1) .or. n>shp(2)) then
         deallocate(Ary)
         allocate(Ary(m, n))
      end if
   else
      allocate(Ary(m, n))
   end if
   Ary = 0.0
end subroutine ReshapeAryD2
```

Figure 2.1: An incarnation of `ReshapeAry`

```
<infonum>
```

and stops. (However, this procedure can be overridden. See §6.1.)

In managing storage, MATRAN always attempts to fit things into existing arrays. Only if the array is too small is it reallocated. The allocation is managed by a generic subroutine `ReshapeAry`. Its function is best illustrated by an example. Figure 2.1 gives an incarnation of this subroutine that reshapes a rectangular double precision array. The arguments `m` and `n` specify the minimal extents of the array. If the array is large enough, the subroutine does nothing, except set the array to zero. If not it deallocates the array, if necessary, allocates it to have shape `(m,n)`, and sets it to zero. The module `MatranUtil_m` provides subroutines to reshape linear and rectangular arrays of type integer, double precision, and double complex.

3. The types Rmat and Rdiag

In this section we will consider the two matrix types currently implemented in MATRAN: the `Rmat` and the `Rdiag`. It is important to keep in mind that a MATRAN matrix type is really a storage type. In particular, the type `Rmat` implements double precision floating-point matrices that can be represented in natural order in a rectangular array. In

```
type Rmat
   real(wp), pointer &                 ! The matrix array
           :: a(:,:) => null()         !
   integer :: nrow = 0                 ! Number of rows in the matrix
   integer :: ncol = 0                 ! Number of columns in the matrix
   integer :: narow = 0                ! Number of rows in the array
   integer :: nacol = 0                ! Number of columns in the array
   character(2) &                      ! Type of matrix
           :: tag = 'GE'               !
   logical :: adjustable =.true.       ! Adjustable array
   integer, pointer                    ! Intermediate value
           :: temporary => null()      !
end type Rmat
```

Figure 3.1: The type Rmat

principle, this means any double precision matrix; but if we add the requirement that the representation use storage efficiently, the set of candidates for a `Rmat` shrinks. For example, a diagonal matrix could be written as a `Rmat`. But that would be an inefficient use of storage, since a diagonal matrix of order n has at most n nonzero elements, all lying on its diagonal. Therefore, MATRAN provides a type `Rdiag` which stores the nonzero elements in a linear array.

3.1. The type Rmat

The type `Rmat` in Figure 3.1 is defined in the module `Rmat_m`. Let us look at the components in order.

• `a(:,:)`. This is the array containing the matrix. It can be allocated and deallocated, so that over time the array of a `Rmat` can vary in size.

The reason for using a single letter `a` for the array of a `Rmat` is that the elements of the matrix are referenced through the array. If `X` is a `Rmat`, then `X%a(i,j)` is the `(i,j)`-element of the corresponding matrix. This is easier to read in a program than a lengthier alternative like `X%Array(i,j)`.

The array `a` of a `Rmat` is always rectangular. This means, as we have noted earlier, that MATRAN has no vector types as such. Instead, an $n\times 1$ matrix represents a column vector and an $1\times n$ matrix represents a row vector.

The initial status of `a` is unassociated. An important convention of MATRAN is the following.

> *If the array of a* `Rmat A` *is associated, then* `A` *is a well-formed* `Rmat`*; i.e.,* `a` *has the dimensions* `narow` *and* `nacol` *and* `0` $\leq$ `nrow` $\leq$ `narow` *and* `0` $\leq$ `ncol` $\leq$ `nacol`. (3.1)

• `nrow, ncol, narow, nacol.` The convention (3.1) shows that MATRAN makes a distinction between a matrix and the array that contains it. The dimensions of the latter can be greater than the former. Thus a `Rmat` must have two pairs of dimensions, one for the matrix and one for the array that contains it. The matrix of a `Rmat` is always in the northwest corner of the corresponding array, and all entries of the array outside the matrix are zero.

It is permissible for `nrow` or `ncol` (or both) to be zero. Such a matrix is called a *null matrix*. Null matrices are especially useful in starting off matrices that expand as an algorithm progresses.

• `tag.` We have already mentioned that `Rmat`s can represent different kinds of commonly used matrices. The `tag` component specifies the kind of matrix, as shown in the following table.

Matrix type	Tag
General	`GE`
Upper triangular	`UT`
Lower triangular	`LT`
Symmetric	`SY`
Symmetric positive (semi) definite	`SP`

The tag of a `Rmat` tells programs that manipulate the `Rmat` that there is special structure present. For example, if the tag of `A` is `UT`, the routine in the Solve suite that computes $A^{-1}B$ uses a special BLAS algorithm to compute its result.

The tags `UT` and `LT` apply to rectangular matrices as well as square ones. In MATRAN, a matrix A, regardless of its dimensions, is upper triangular if

$$i > j \implies a_{ij} = 0$$

and is lower triangular if

$$i < j \implies a_{ij} = 0.$$

Rectangular triangular matrices are sometimes called *trapezoidal* in the literature.

Matrices with the tag `SP` are usually generated in a way that mathematically guarantees that they are positive definite, or at least positive semidefinite (e.g., as with the cross-product $A^{\mathrm{T}}A$). However, it should be kept in mind that rounding error may cause the matrix to not be definite. In such cases the constructor for the Cholesky decomposition will fail See §6.3).

MATRAN does not support packed versions of the matrices in the table above. Thus an upper triangular matrix is represented in a rectangular array zeros and all. So that everyone is sure what is in the array of a `Rmat`, we adopt the following convention.

> *A matrices is fully represented in the array of its* `Rmat`. *Elements of the array outside the matrix are zero.*

Thus, in a symmetric `Rmat` both the upper and lower part of the matrix are present.[5]

• `adjustable`. This component addresses the following problem. It may sometimes happen that a result to be stored in a `Rmat` is larger than the array of the `Rmat`. If the `Rmat` is `adjustable`, then MATRAN is permitted to reallocate the array to contain the result. We will return to this point at the end of this section.

• `temporary`. This component is used in conjunction with `SetTemp`, `GuardTemp`, and `CleanTemp` to deallocate temporary `Rmats`. If `temporary` is `null()`, the `Rmat` is not temporary. If `temporary` is one or greater the `Rmat` is temporary. As long as you follow the conventions (1.3) and (1.4), your temporary arrays will be deallocated at the proper time. Note that `temporary` should be manipulated only by `SetTemp`, `GuardTemp`, and `CleanTemp`.[6]

—

As mentioned above, he module `Rmat_m` defines the three generic subroutines `SetTemp`, `GuardTemp`, and `CleanTemp` used to deallocate temporaries. It also defines a sanitizer `Clean` that restores a `Rmat` to its pristine condition.

The module `Rmat_m` overloads the assignment operator for `Rmats` in four ways.

`Rmat A = Rmat B`

> The statement `A = B` copies `B` to `A`. It is not quite an exact copy: `A%temporary` and `A%adjustable` are unchanged whatever the values of the corresponding components of `B`. Moreover, the shape of `A%a` may be different from `B%a`, as we will see in a moment.

[5] All this is consistent with the fact that MATRAN segregates matrices by storage type. A packed symmetric matrix, for an example, would be a new storage type and would have to have its own defined type.

[6] For those who want the full story, here it is. The real problem with temporaries is knowing when to deallocate them. If, for example, a subprogram with a temporary argument passes it on to another subprogram, the second subprogram should not deallocate it, since the invoking program may need to use it on return. To avoid premature deallocation, `GuardTemp` simply increases `temporary` by one, provided it is nonnull. `CleanTemp` decreases `temporary` by one provided it is greater than one, but it does not deallocate the array `a` unless `temporary` is one after decrementation. You can easily convince yourself that if the convention (1.3) is followed religiously then only the first subprogram invoked with the temporary `Rmat` will deallocate its storage.

`Rmat A = Array B(:,:)`
: The statement `A = B` causes `A` to be a `Rmat` whose matrix is the contents of `B`. `A%tag` is set to `GE`, The components `A%temporary A%adjustable` remain unchanged.

`Rmat A = integer vec(:)`
: If `vec = (/m,n/)`, then `A` becomes an `m`x`n` zero matrix an an array whose size is determined as described below. If `vec = (/m,n, ma,na/)`, then `A` becomes an `m`x`n` zero matrix contained in an `ma`x`na` array. The component `adjustable` remains unchanged, but the array will be adjusted, whether or not the `Rmat A` is adjustable. The array `A%a` is set to zero. The array `A%temporary` is unchanged.

`Rmat A = real(wp) s`
: The statement `A = s` produces a `1`x`1` `Rmat` whose single element is `s`.

Three of these overloaded assignments have operator forms, generically written `.dm.`, for use in expressions.

`.dm.ary`
: Produces a `Rmat C` defined by `C = ary`, where `ary` is a rectangular array.

`.dm.vec`
: Produces a `Rmat C` defined by `C = vec`, where `vec=(/m,n/)` or `vec=(/m,n, ma,na/)`.

`.dm.s`
: Produces a `Rmat C` defined by `C = s`, where `s` is of type `real(wp)`.

The `Rmats` created by `Rmat A = vec` and `.dm.vec` are initialized to zero. Hence MATRAN does not provide special routines to construct zero matrices.

It is now time to be more precise about how MATRAN treats arrays. When MATRAN must transfer an `m`x`n` matrix to a `Rmat A`, it always tries to use the space available in `A%a`. If `A%a` can contain the matrix MATRAN uses `A%a` as is. If `A%a` is too small and `A` is adjustable, MATRAN reallocates `A%a` to be `m`x`n`. Otherwise, MATRAN gives an error return. A good way of summing this up is to say: *Left to itself* MATRAN *may increase the size of a* `Rmat` *array, but it will not decrease it.* The only exceptions are the subroutine `Clean`, which deallocates the array, and the assignment `Rmat = vec` which changes the array shape according to the contents of `vec`.

The above recipe for adjusting arrays is implemented by the generic subroutine

```
subroutine ReshapeAry(A, n, m)
```

Here `m` and `n` are the row and column dimensions of the matrix to be placed in `A`. The final array is always set to zero. We have already seen an example of this subroutine in

```
   interface assignment (=)
      module procedure RmEqualsRm, RmEqualsAry, RmEqualsRowCol
   end interface
   ...
contains
   ...
   subroutine RmEqualsRm(A, B)
      type(Rmat), intent(inout) :: A
      type(Rmat), intent(in) :: B

      call GuardTemp(B)

      call ReshapeAry(A, B%nrow, B%ncol)

      A%a = 0
      A%a(1:A%nrow, 1:A%ncol) = B%a(1:B%nrow,1:B%ncol)
      A%tag = B%tag

      call CleanTemp(B)

   end subroutine RmEqualsRm
   ...
```

Figure 3.2: Implementation of `Rmat = Rmat`

Figure 2.1, where the concern was with reshaping a raw array, rather than the array of a matrix type.

We conclude this subsection with the implementation in Figure 3.2 of the assignment `Rmat = Rmat`, which illustrates some of the points above. Many of the subprograms implementing MATRAN are as simple as this. When in doubt about what MATRAN does in a particular situation, try looking at the code.

3.2. The type Rdiag

The type `Rdiag` implements a diagonal matrix. It is defined in the module `Rdiag_m` by

```
type Rdiag
   real(wp), pointer &            ! The matrix array
        :: a(:) => null()
   integer :: order = 0           ! The order of the matrix
   integer :: na = 0              ! The length of the array
   logical :: adjustable = .true. ! Adjustable array
   integer, pointer&              ! Intermediate value
        :: temporary => null()
end type Rdiag
```

The components of `Rdiag` are analogous to those of `Rmat`.

• `a(:)`. Since a diagonal matrix is nonzero only on its principal diagonal, it can be represented by a linear array, which in a `Rdiag`, as with a `Rmat`, is called `a`.

• `order, na`. The `order` of the diagonal matrix represented by a `Rdiag` can be less than the size `na` of the array containing its diagonal.

• `adjustable, temporary`. These components serve the same functions as they do in a `Rmat`.

—

The module `Rdiag_m` defines the usual generic subroutines `SetTemp`, `GuardTemp`, and `CleanTemp` for dealing with temporaries. It also defines `ReshapeAry`, whose calling sequence is

```
call ReshapeAry(Rdiag, n)
```

to reallocate the array `a`, if necessary. As with a `Rmat`, `Clean(D)` restores the `Rdiag D` to its default state.

`Rdiag_m` also overloads the assignment operator. The implementing functions all use `ReshapeAry` to get storage. The components `temporary` and `adjustable` are unchanged.

`Rdiag D = Rdiag E`
: The statement `D = E` copies `E` to `D`.

`Rdiag D = Array E()`
: The statement `D = E` causes `D` to be a `Rmat`, whose diagonal is the contents of `E`. The component `adjustable` remains unchanged.

`Rdiag D = vec`
: If `vec = (/n/)`, then `D` is a zero `Rdiag` of order `n` in an array obtained by reshaping `D%a`. If `vec = (/n, na/)` then `D` is a zero `Rdiag` of order `n` in an array of length `na`. Note that the array will be adjusted regardless of the status of the component `adjustable`, which remains unchanged.

`Rdiag D = real(wp) s`
: The statement `D = x` produces a `Rdiag` of order one whose single diagonal element is `s`.

`Rmat A = Rdiag D`
: `A` is the `Rmat` corresponding to `D`.

Note that there is no operator corresponding to `Rdiag D = Rmat A` to extract the diagonal of a Rmat. See the `RmatDiag` suite.

The `Rdiag` suite also has conversion operators.

`.dd.ary`
: Produces a `Rdiag D` defined by `D = ary`, where `ary` is a linear array.

`.dd.vec`
: Produces a `Rdiag D` defined by `D = vec`, where `vec = (/order/)` or `vec = (/order, na/)`.

`.dd.s`
: Produces a `Rdiag D` defined by `D = s`, where `s` is of type `real(wp)`.

`.dm.D`
: Produces a `Rmat A` defined by `A = D`, where `D` is a `Rdiag`.

4. Matrix Operations

In this section we introduce the basic matrix operations supported by MATRAN. Other, less basic operations are gathered together in a loose grab bag called matrix miscellany.

4.1. Generalities

Matrix operations in MATRAN are divided into *suites* of related generic subroutines and operators. Here is a list of the operator suites described in this section.

Transpose	A^{H}, A^{T}
Sum	$A+B$, $A-B$, $-A$
Product	σA, AB, $A^{\mathrm{T}}B$, $\ldots$
Solve	$A^{-1}B$, AB^{-1}, $A^{-\mathrm{T}}B$, $\ldots$
Join	$(A\ B)$, $\begin{pmatrix} A \\ B \end{pmatrix}$
Border	$A = (A\ B)$, $A = (B\ A)$, $\ldots$
Submatrix	$A(i_1{:}i_2, j_1{:}j_2)$, $A(:,j)$, $\ldots$

Each suite is implemented by a sequence of modules corresponding to the derived matrix types in the wrapper. The types are arranged in a hierarchy, and each module is responsible for providing operations for both its type and for types lower in the hierarchy.

For example, suppose MATRAN has three types, `Rmat`, `Rdiag`, and `Cmat`, arranged hierarchically in that order. Then the module `RmatSum_m` is responsible for all sum operations between `Rmat`s. The module `RdiagSum_m` is responsible for all sum operations between `Rdiag`s and `Rdiag`s and `Rmat`s. `CmatSum_m` is responsible for all sums between `Cmat`s and `Cmat`s, `Rdiag`s, and `Rmat`s.

In addition the type that is higher in the hierarchy has the responsibility for implementing mixed assignment operators involving itself and types lower in the hierarchy. That is why the assignment `Rmat = Rdiag` is implemented in `Rdiag_m` instead of `Rmat_m`.

This system has the advantage of clearly delineating who is responsible for what, so that it is conceptually easy to add new types to the wrapper. However, the code needed to implement a new type grows at least quadratically with the number of types. Fortunately, it may not be necessary to implement all possible combinations of operations. For example, if someone decides to introduce a type `Dband` for band matrices, it may be decided that while we need a product between `Dband`s and `Rmat`s, we do not need a product between `Dband`s and `Dband`s.

Except for the Border suite, matrix operations are implemented in two forms: as an operator (or function) and as a subroutine. For example, the `*` operator is overloaded so that the expression

```
C = A*B                                                                    (4.1)
```

results in a `Rmat` `C` containing the product of the matrices A and B. This is the form one would ordinarily use. However, it has some hidden storage allocation in the form of a temporary `Rmat` to hold the product `A*B` before it is assigned to `C`.

Temporary objects are a potential source of inefficiency, since in a loop they are repeatedly allocated and deallocated. For programs involving large matrices this will not usually be a problem; the arithmetic calculations will tend to dominate. For small matrices, however, calls to the allocator may slow things down. To address this problem MATRAN shadows each operation with a subroutine that performs the operation and places the result in a `Rmat` of your choosing. Suppose, for example, we have a loop of the form

```
do i=1, maxi
   ...
   r = b - A*x
   ...
end do
```

If we make the declaration

```
type(Rmat) :: temp
```

Operation	Operator	Subroutine
$C = A^{\mathrm{H}}$	`C = .ctp.A`	`call Ctp(C, A)`
$C = A^{\mathrm{T}}$	`C = .trp.A`	`call Trp(C, A)`

- These operations are not available for `Rdiags`

Figure 4.1: The Transpose Suite

then we can write

```
do i=1,maxi
   ...
   call Times(temp, A, x)
   call Minus(r, b, temp)
   ...
end do
```

This does not get rid of the need for the temporary `temp` to hold the intermediate value `A*x`, but `temp`'s storage is reused rather than being allocated and deallocated with each iteration of the loop.

It is recommended that one initially use operators to write and debug programs, after which they can be fine tuned by using the subroutine forms where necessary.

4.2. The Transpose suite

The Transpose suite has two operations: the conjugate transpose and the transpose, as given in Figure 4.1. The format of the table is the desired matrix operation, the operator version, and the subroutine version.

We have already observed that defined binary operators bind so loosely that it may be necessary to use parentheses to make an expression parse correctly. The operators in this suite are unary operators. By Fortran 95 convention they have precedence over all other operators. Thus `A + .cpt.B` does not have to be recast in the form `A + (.ctp.B)` to work as expected.

It is important to note that for real matrices the transpose and the conjugate transpose are the same. It is strongly recommended that the conjugate transpose be used in working with real matrices. In the overwhelming majority of cases, when a program dealing with real matrices is rewritten for complex matrices, the conjugate transpose is what you want. The transpose operator should be used exclusive with complex matrices.

This convention affects the nomenclature of some of MATRAN's operations. For example, for real matrices the operator that computes $A^{\mathrm{T}}B$ is `.xhy.`, not `.xty.` as might be expected. See the Product and Solve suites.

Operation	Operator	Subroutine
$C = A + B$	`C = A + B`	`call Plus(C, A, B)`
$C = A - B$	`C = A - B`	`call Minus(C, A, B)`
$C = -A$	`C = -A`	`call Minus(C, A)`

- These operations are defined for any combination of `Rmats` and `Rdiags`.

Figure 4.2: The Sum suite

4.3. The Sum suite

The Sum suite overloads the operators `+` and `-` to provide the sum and difference of two matrix objects. In addition the suite implements the unary minus. Figure 4.2 shows the usage.

The operations set the tags of the results appropriately. For example if `A` and `B` are flagged `UT`, so is `C`. The other suites do the same.

4.4. The Product suite

The product suite implements products of matrices and their transposes, as shown in Figure 4.3

All the operations in the suite involving transposes could be implemented using the operator `*` and `.ctp.` operator from the Transpose suite. For example, to compute $C = A^{\mathrm{H}}B$ one could write

```
C = .ctp.A*B
```

where `.ctp.` is the MATRAN unary operator that computes the conjugate transpose (the same as the transpose for real matrices). However, one can also write

```
C = A.xhy.B
```

where by convention `xhy` is shorthand for $X^{\mathrm{H}}Y$. The second form is superior to the first, since the second calls a BLAS subroutine that calculates $A^{\mathrm{H}}B$ directly from A and B without forming the transpose.

The `Rmats` produced by `.xhx.` and `.xxh.` are tagged `SP`. Mathematically, these matrices have to be at least semidefinite; however, rounding error may cause the computed matrices to be indefinite.

Ordinarily, the operands in a product must be conformable for matrix multiplication — that is, the number of columns of the first operand must be the same as the number rows of the second. However, if one of the operands represents a 1×1 matrix,

Operation	Operator	Subroutine
$C = sA$	`C = s*A`	`call Times(C, s, A)`
$C = As$	`C = A*s`	`call Times(C, s, A)`
$C = AB$	`C = A*B`	`call Times(C, A, B)`
$C = A^{\mathrm{H}}B$	`C = A.xhy.B`	`call TimesXhy(C, A, B)`
$C = AB^{\mathrm{H}}$	`C = A.xyh.B`	`call TimesXyh(C, A, B)`
$C = A^{\mathrm{H}}A$	`C = .xhx.A`	`call TimesXhx(C, A)`
$C = AA^{\mathrm{H}}$	`C = .xxh.A`	`call TimesXxh(C, A)`

- In the above `s` is a scalar.
- The operations `s*A`, `A*s`, and `A*B` are defined for any combinations of `Rmat`s and `Rdiag`s.
- The operations `A.xhy.B`, `A.xyh.B`, `.xhx.A`, and `.xxh.A` are defined for `Rmat`s only.

Figure 4.3: The Product suite

which is essentially a scalar, this requirement is dropped. A common example of this is the statement

```
xp = x - (q.xhy.x)*q
```

which orthogonalizes the vector `x` against the vector `q` of 2-norm one.[7]

4.5. The Solve suite

The Solve suite contains operations to compute the product of a matrix and its inverse. It is called the Solve suite, because a principal application is to solve linear systems like $Ax = b$, whose solution can be written in the form $x = A^{-1}b$. The routines do not compute an inverse and multiply; instead, if necessary, they computed a decomposition of the matrix in question and use it to solve systems of equations to get the answer.

The operations are shown in Figure 4.4. These operations interrogate the tag field of the `Rmat` whose inverse appears in the first column. If the matrix is triangular, it solves the system directly using an appropriate BLAS. If not, it computes a pivoted LU decomposition (`tag = GE, SY`) or a Cholesky factor (`tag = SP`) and uses that to perform the operation.

[7] At least mathematically. Numerically, `xp` and `x` may be far from orthogonal. A way out of this predicament is given by the subroutine `gsro` in §8.

Operation	Operator	Subroutine
$C = A/s$	`C = A/s`	`call Solve(C, A, s)`
$C = A^{-1}B$	`C = A.xiy.B`	`call SolveXiy(C, A, B)`
$C = A^{-\mathrm{H}}B$	`C = A.xihy.B`	`call SolveXihy(C, A, B)`
$C = AB^{-1}$	`C = A.xyi.B`	`call SolveXyi(C, A, B)`
$C = AB^{-\mathrm{H}}$	`C = A.xyih.B`	`call SolveXyih(C, A, B)`

- Except as noted below, these operations are defined for `Rmat`s and `Rdiag`s.
- The operations `A.xihy.B` and `A.xyih.B` are defined only for `Rmat`s.
- The operation `A.xiy.B` is not defined for `A` a `Rmat` and `B` a `Rdiag`. Use the Inverse suite.
- The operation `A.xyi.B` is not defined for `A` a `Rdiag` and `B` a `Rmat`. Use the Inverse suite.

Figure 4.4: The Solve suite

In many applications, systems involving the same matrix must be solved repeatedly. For matrices of tag `GE, SY`, or `SP`, this means recomputing a factorization of the same matrix for each solve operation. To avoid this expense, the solve subroutines have two additional optional arguments `LU` and `Chold`. To see how this is used, consider the following code

```
do
   call SolveXiy(y, A, x, LU=lua)
   ...
   <modify x>
   ...
end do
```

At each call, `SolveXiy` determines if `LU` contains a pivoted LU decomposition by checking its `companion` component. If it does does not, then `SolveXiy` initializes `LU` to an LU decomposition of `A`. Otherwise, `SolveXiy` assumes that the LU decomposition is associated of `A`. In either case, it uses the LU decomposition to compute $A^{-1}x$. It is the responsibility of the user to maintain the integrity of the relation between `A` and `LU`. The programmer can announce that the relation has been broken by setting (in the above example)

```
lua%companion = .false.
```

in which case `SolveXiy` will compute a new factorization.

Operation	Operator	Subroutine
$C = (A\ \ B)$	`C = A.jwe.B`	`call JoinWE(C, A, B)`
$C = \begin{pmatrix} A \\ B \end{pmatrix}$	`C = A.jns.B`	`call JoinNS(C, A, B)`

- These operations are defined for any combinations of `Rmats` and `Rdiags`.

Figure 4.5: The Join suite

4.6. The Join suit

Given two matrices A and B we can join them in two ways. First, if A and B have the same number of rows, we can form the matrix

$$C = (A\ \ B).$$

We say that A and B have been joined from west to east. Second, if the two matrices have the same number of columns we can form the matrix

$$C = \begin{pmatrix} A \\ B \end{pmatrix}.$$

We say that the matrices have been joined north to south.

MATRAN's Join suite provides these operation, as shown in Figure 4.5.

4.7. The Border suit

Many matrix algorithms expand a matrix by bordering it with other matrices. For example, we might replace A with

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

This bordering can be implemented using the Join suite by the following fragment.

```
A = A.jwe.B
T = C.jwe.D                                    (4.2)
A = A.jns.T
```

However, this code is awkward and requires four temporaries — three implicit temporaries for the assignments and the explicit temporary `T`. MATRAN allows you to accomplish this by a single call to a subroutine:

Operation	Subroutine	Result
Border southeast	`BorderSE(A, S, E, SE)`	`[A, E; S, SE]`
Border northeast	`BorderSE(A, N, E, NE)`	`[N, NE; A, E]`
Border northwest	`BorderNW(A, N, W, NW)`	`[NW, N; W, A]`
Border southwest	`BorderNW(A, S, W, SW)`	`[W, A; SW, S]`
Border north	`BorderN(A, N)`	`[N; A]`
Border south	`BorderS(A, S)`	`[A; S]`
Border east	`BorderE(A, E)`	`[A, E]`
Border west	`BorderW(A, W)`	`[W, A]`

- The result is expressed in MATLAB notation.
- All arguments to a Border subroutine must be of the type `Rmat`.

Figure 4.6: The Border suite

```
call BorderSE(A, C, B, D)
```

Since there are many ways of bordering, let us introduce some conventions. In the above example, we say that `A` is border on the southeast. Obviously, we can also border on the southwest, the northeast, and the northwest. Moreover, we can border A by a single matrix to the north, south, east and west. Figure 4.6 describes the subroutines that accomplish the bordering.

Arguments in the border subroutines must have dimensions for which the operation make sense. For example in `BorderE(A, E)` both `A` and `E` must have the same number of rows.

The subroutines of the Border suite are generic and could potentially mix matrix types. However, the number of arguments to the border subroutines is so great that we would have an explosion of implementing subroutines. For example if we allow arbitrary combinations of `Rmat`s and `Rdiag`s, the suite would have 264 subroutines. For this reason, MATRAN allows only matrices of a single type in the arguments of a border subroutine — and at present that is only the type `Rmat`. One cure for the problem of mixed types is to convert every argument of the function to the the type of the natural result before calling the subroutine. Another is to use the Join suite, which does allow mixed types. See (4.2). Fortunately, mixed types are rare in practice.

4.8. The Submatrix suite

The final suite extracts submatrices from a `Rmat`. Since specifying a submatrix requires information beyond the `Rmat` in question, submatrix extraction cannot be implemented

Submatrix	Function	Subroutine
$C = A(i_1{:}i_2, j_1{:}j_2)$	`C = Sbm(A, i1, i2, j1, j2)`	`GetSbm(C, A, i1, i2, j1, j2)`
$C = A(:, j_1{:}j_2)$	`C = Col(A, j1, j2)`	`GetCol(C, A, j1, j2)`
$C = A(:, j)$	`C = Col(A, j)`	`GetCol(C, A, j)`
$C = A(i_1{:}i_2, :)$	`C = Row(A, i1, i2)`	`GetRow(C, A, i1, i2)`
$C = A(i, :)$	`C = Row(A, i)`	`GetRow(C, A, i)`

- These routines are defined only for `Rmat`s.

Figure 4.7: The Submatrix suite

as a defined operator. Instead, we give functions and companion subroutines, as shown in Figure 4.7.

The effect of these functions can also be attained by using the operator .dm. combined with Fortran 95's subarray expressions. For example `Sbm(A, i1, i2, j1, j2)` is equivalent to `.dm.A%a(i1:i2, j1:j2)`. However, one must be careful with colons. The equivalent of $Col(A, j)$ is `.dm.A%a(1:A%nrow ,j)`, not `.dm.A%a(:, j)`.

5. Matrix miscellania

This section describes a miscellany of suites to perform various functions. Right now it is rather small, but it will grow.

5.1. The Diag suite

The kth diagonal $\mathrm{diag}(A, k)$ of a matrix A is defined as the diagonal starting with $a_{1,k+1}$ if $k \geq 0$ and with $a_{-k+1,1}$ if $k < 0$. Thus $\mathrm{diag}(A, 0)$ is the main diagonal of A; $\mathrm{diag}(A, 1)$ is the first superdiagonal; and $\mathrm{diag}(A, -1)$ is the first subdiagonal. The Diag suite provides a generic subroutine and function for extracting a diagonal.

The subroutine has the form

```
subroutine GetDiag(D, A, k)
```

where

`type(Rdiag), intent(inout) :: D`
: On return contains the `k`th diagonal of `A`.

`type(Rmat), intent(in) :: A`
: The matrix whose diagonal is to be extracted.

`integer, optional, intent(in) :: k`
: The diagonal to be extracted. If not present, extract the main diagonal.

The function has the form

```
function Diag(A, k) result(D)
```

where

`type(Rdiag) :: D`
  A `Rdiag` containing on return the `k`th diagonal of `A`.

`type(Rmat), intent(in) :: A`
  The matrix whose diagonal is to be extracted.

`integer, optional, intent(in) :: k`
  The diagonal to be extracted. If not present, extract the main diagonal.

5.2. The Eye suite

The module `RmatEye_m` generates identity matrices — or rather zero matrices with ones on their principal diagonals. As usual, it defines both a generic subroutine and an associated function. The subroutine has the calling sequence

```
call Eye(A, m, n)
```

where

`type(Rmat), intent(inout) :: A`
  On return A is a `Rmat` with ones on its principal diagonal and zeros elsewhere.

`integer :: m`

`integer, optional :: n`
  If `n` is not present `A` is `m` x,`m`.
  If `n` is present `A` is `m` x `n`.

The functional form is

```
function Reye(m, n) result(A),
```

where

`type(Rmat) :: A`
  On return `I` is a `Rmat` with ones on its diagonal and zeros elsewhere.

`integer :: m`

`integer, optional :: n`
  If `n` is not present `A` is `m` x,`m`.
  If `n` is present `A` is `m` x `n`.

5.3. The Inverse suite

The inverse of a matrix is seldom needed: the Solve suite computes matrices like $A^{-1}B$ faster and more stably than by inverting A and multiplying. But for the rare occasions when an explicit inverse is required, MATRAN provides the Inverse suite. As usual it has a subroutine and operator form.

The subroutine has the form

```
subroutine Inv(C, A, luda, chola, info, mywork)
```

where

`type(Rmat), intent(out) :: C`
: The inverse matrix.

`type(Rmat), intent(in) :: A`
: The matrix to be inverted.

`type(RmatLudpp), optional, intent(inout) :: luda`
: A pivoted LU decomposition. If present and `luda.companion` is true, the decomposition is used to compute the inverse. If present and `luda.companion` is false, the LU decomposition is computed and returned. If absence an LU decomposition is silently computed. Applies only to `Rmat`s of tag GE.

`type(RmatChol), optional, intent(inout) :: chola`
: A Cholesky decomposition. If present and `luda.compantion` is true, the decomposition is used to compute the inverse. If present and `chola.companion` is false, the decomposition is computed and returned. If absence a Cholesky decomposition is silently computed. Applies only to `Rmat`s of tag SP.

`integer, optional, intent(out) :: info`
: When a decomposition is computed to calculated the inverse, `info`, if present, contains on return the value of the `info` parameter from the LAPACK routine that computed the decomposition. Applies only to `Rmat`s of type `GE`, `SY`, and `SP`.

`real(wp), pointer :: mywork(:)`
: For matrices of type `SY`, the LAPACK routine `DSYTRF` requires an auxiliary work array, which is ordinarily allocated and deallocated by `Inv`. If `mywork` is present and contains enough storage, it is used as the work array. If it is present but does not contain enough storage, it is reallocated and used as the work array. This storage is not deallocated, so that `mywork` can be reused when `Inv` is called in a loop.

The operator has the form

```
.inv.A
```

where `A` is a `Rmat`.

5.4. The Norm and Norm2 suites

The Norm suite provides generic functions to compute the following three norms.

1. The 1-norm: $\|A\|_1 = \max_j \sum_i |a_{ij}|$
2. The Frobenius norm: $\|A\|_{\mathrm{F}} = \sqrt{\sum_{ij} |a_{ij}|^2}$
3. The ∞-norm: $\|A\|_\infty = \max_i \sum_j |a_{ij}|$

The functions have the following calling sequences.

```
Norm1(A);   NormF(A);   NormInf(A)
```

where `A` is a `Rmat`.

The 2-norm of a matrix A is defined by

$$\|A\|_2 = \max_{\|x\|_{\mathrm{F}}=1} \|Ax\|_{\mathrm{F}}.$$

The Norm2 suite provides a generic function

```
Norm2(A)
```

to compute the 2-norm of a `Rmat`. The reason that the 2-norm is segregated in a separate suite is that its computation requires the expensive solution of an eigenvalue problem. Think twice before using it!

5.5. The Pivot suite

The Pivot suite provides subroutines to apply interchanges to the rows or columns of a `Rmat`, thus effecting a permutation of the rows or columns. It also applies the inverse permutation. The permutation is specified by an array `pvt` of length `npvt`. The effect of pivoting and its inverse on an array x is given by the following fragments of pseudo-code.

```
Pivoting                              Inverse pivoting

do i=1 to npvt                        do i=npvt,1,-1
   swap x(i) and x(pvt(i))               swap x(i) and x(pvt(i))
end do                                end do
```

There are four generic functions in the suite.

```
subroutine PivotRow(A, pvt, npvt)

subroutine PivotInvRow(A, pvt, npvt)

subroutine PivotCol(A, pvt, npvt)

subroutine PivotInvCol(A, pvt, npvt)
```

where

`type(Rmat), intent(inout) ::  A`
  The `Rmat` to be pivoted

`integer, intent(in) ::  pvt(:)`
  The pivot array

`integer, intent(in) ::  npvt`
  The number of pivots.

In the names of these subroutines, `Row` indicates that the rows of `A` are interchanged, `Col` that the columns of `A` are interchanged, and `Inv` that the inverse pivoting is performed.

5.6. The Print suite

Fortran 95 has the ability to print objects in any conceivable format, and it is expected that most programmers will wish to custom code their output. However, in debugging MATRAN code, it is convenient to be able to print out `Rmat`s and their arrays in a standard format. The Print suit provides a generic subroutine to do this.

The subroutine to print a rectangular array has the calling sequence

```
call Print(A, m, n, w, d, e, lw, nbl)
```

where

`real(wp), intent(in) ::  A(:,:)`
  The array to be printed.

`integer, intent(in) ::  m`
  The number of rows to print.

`integer, intent(in) ::  n`
  The number of columns to print.

`integer, intent(in) ::  w`

`integer, intent(in) ::  d`

`integer, optional, intent(in) :: e`
: This and the next two argument specify the format by which the elements are to be printed. Specifically, the elements are printed in 1pe<w>.<d>e<e> format. The exponent width field e is optional. Its default value is 3.

`integer, optional, intent(in) :: lw`
: The width in characters of an output line. The default value is 80.

`logical, optional, intent(in) :: nbl`
: If nbl (for no blank line) is present and true, it suppresses the printing of a blank line above the array.

The subroutine to print a Rmat has the calling sequence.

```
call Print(A, w, d, note, e, lw)
```

where

`type(Rmat), intent(in) :: A`
: The Rmat that is to be printed.

`integer, intent(in) :: w`

`integer, intent(in) :: d`

`integer, optional, intent(in) :: e`
: This and the next two argument specify the format by which the elements are to be printed. Specifically, the elements are printed in 1pe<w>.<d>e<e> format. The exponent width field e is optional. Its default value is 3.

`character(*), optional, intent(in) :: note`
: If present the string note is printed along with the array.

`integer, optional, intent(in) :: lw`
: The width in characters of an output line. The default value is 80.

This print function also prints

```
A%nrow, A%ncol, A%narow, A%nacol, A%tag, A%adjustable, A%temporary
```

(Actually, Print tells a little white lie. If pointer `A%temporary` is associated it prints the value of its target; if not, it prints zero.) Here is some sample output generated by

```
call Print(A, 9, 1, 'This is the Rmat A')
```

```
This is the Rmat A
4 5 4 5 GE T 0
          1         2         3         4         5
1  2.0E+000 3.0E+000 4.0E+000 5.0E+000 6.0E+000
          1         2         3         4         5
2  3.0E+000 4.0E+000 5.0E+000 6.0E+000 7.0E+000
          1         2         3         4         5
3  4.0E+000 5.0E+000 6.0E+000 7.0E+000 8.0E+000
          1         2         3         4         5
4  5.0E+000 6.0E+000 7.0E+000 8.0E+000 9.0E+000
```

5.7. The Rand suite

MATRAN provides routines to generate uniformly or normally distributed random `Rmat`s. There are two subroutine forms.

```
call RandX(A, m, n)
```

where `X` is either `U` or `N`. If `X = U` the elements of the matrix are independently uniformly distributed in $[0, 1)$. If `X = N` the elements of the matrix are independently normally distributed $(0, 1)$.

`type(Rmat), intent(inout) :: A`
: The random `Rmat` generated by the subroutine.

`integer, intent(in) :: m`

`integer, optional, intent(in) :: n`
: If `m` is not present, `A` is `m` x `m`. If `m` is present, `A` is `m` x `n`.

The functional forms are

```
DrandX(m, n) result(A)
```

where `X` is the suffix `U` or `N`, as described above, and

`type(Rmat), intent(inout) :: A`
: The random `Rmat` generated by the subroutine.

`integer, intent(in) :: m`

`integer, optional, intent(in) :: n`
: If `n` is not present, `A` is `m` x `m`. If `m` is present, `A` is `m` x `n`.

The uniformly distributed random variables are obtained using the Fortran 95 intrinsic subroutine `random_number`, and the user is warned that the quality of the pseudorandom numbers so generated are implementation dependent. Normally distributed random numbers are computed by an algorithm of Leva [ACM Trans. Math. Software, 18 (1992) 454–455.]

To control the seed for both uniform and normal random matrices, use the intrinsic subroutine `random_seed`.

6. Decompositions

6.1. Generalities

A matrix decomposition is a factorizations of a matrix into a product of two or more matrices. MATRAN provides a number of standard decompositions. The factors of each decomposition are generated by a generic subroutine, which puts the factors in a defined type particular to the decomposition, which we will call the container of the decomposition. Here is a list of the decompositions currently provided by MATRAN.

Decomposition	Container	Constructor
LU with partial pivoting	`RmatLudpp`	`Ludpp`
Cholesky decomposition	`RmatChol`	`Chol`
QR decomposition	`RmatQR`	`QR`
QR decomposition with pivoting	`RmatQRP`	`QRP`
Spectral decomposition	`RmatSpec`	`Spec`
Singular value decomposition	`RmatSVD`	`SVD`
Eigendecomposition	`RmatEig`	`Eig`

In addition each decomposition has a generic sanitizer `Clean` to deallocate the storage of decompositions constructed in subprograms.

The standard calling sequence for the constructors is

```
call <constructor>(<container>, <matrix>, <optional arguments>)
```

In order to interact with the LAPACK drivers that compute the decompositions, most of the constructors have optional arguments, in addition to the container and matrix. They fall into two classes.

First, some of the drivers have a parameter called `info` that returns information about the status of the computation. If the status indicates an error, the constructor causes an error message to be printed and terminates the run. However, if the optional parameter `info` is present in the calling sequence of the constructor, the constructor sets it to the value of returned by the driver and returns, thus giving the calling program a chance act on the error flag.

Second, many of the drivers require that the user furnish additional work arrays. Ordinarily, MATRAN silently allocates and deallocates this storage. However, through the optional parameters the user can furnish the working storage explicitly. This may reduce storage management time when the constructor is called inside a loop.

The containers are all derived types — a different one for each decomposition. But they all have a common component `companion` that is used to control the reuse of a decomposition. Specifically, consider the following loop

```
do
  call Ludpp(lua, A)
  <calculations involving lua>
  if (<condition>) then
     <modify A>
  end
end do
```

Suppose that the `if` statement is only place in the loop where `A` is modified. Then if `<condition>` is not true the call to `Ludpp` is redundant — expensively redundant. To cure this problem we can code as follows.

```
do
  if (.not.lua%companion)&
     call Ludpp(lua, A)
  <calculations involving lua>
  if (<condition>) then
     <modify A>
     lua%companion = .false.
  end
end do
```

Thus `companion` is a flag that tells the program that a decomposition is associated with a matrix of interest.

In using `companion`, it is important to keep in mind that it does not in itself suppress the computation of the decomposition. It has absolutely no effect on `Ludpp` or any other decomposition constructor. It is just a handy flag that enables the programmer to decide whether or not to compute the decomposition in question.

The default value of `companion` is false. All decomposition constructors set `companion` equal to true.

In the Solve suite we gave an example of the use of `companion` to force the recomputation of a decomposition. The same treatment has been applied to our introductory example `qrlsq` in Figure 6.1. It is worth pondering a bit.

6.2. The LU decomposition

Given an $m\times n$ matrix A, there is a permutation matrix P such that

$$PA = LU, \tag{6.1}$$

```
subroutine qrlsq(A, b, x, r, oldqra)

use MatranRealCore_m
use RmatQR_m
implicit none

   type(Rmat), intent(in) :: A, b
   type(Rmat), intent(out) :: x, r
   type(RmatQr), optional, intent(inout), target :: oldqra

   ! Internal variables.

   type(RmatQR), target :: newqra
   type(RmatQR), pointer :: qra

   !Protect temporaries.

   call GuardTemp(A); call GuardTemp(b)

   ! Get the QR decomposition of A.

   if (present(oldqra)) then
      qra => oldqra
      if (.not.qra%companion) call QR(qra, A)
   else
      qra => newqra
      call QR(qra, A)
   end if

   ! Solve the least squares problem.

   x = qra%R.xiy.(qra%Q.xhy.b)
   r = b - A*x

   ! Clean up.

   if (.not.present(oldqra)) call Clean(qra)
   call CleanTemp(A); call CleanTemp(b)

end subroutine qrlsq
```

Figure 6.1: QR least squares

where U is an upper triangular matrix and L is a lower triangular matrix with ones on its diagonal and with its subdiagonal elements not greater than one in magnitude. MATRAN represents such a decomposition by the derived type

```
type RmatLudpp
   type(Rmat) :: L                ! The L-factor
   type(Rmat) :: U                ! The U-factor
   integer, pointer :: pvt(:)     ! The pivot arry
   integer :: npvt                ! The number of pivots.
   logical :: companion           ! True if the decomposition is
                                  ! that of a Rmat of interest.
end type RmatLudpp
```

The members `L` and `U` are `Rmats` with flags `LT` and `UT` respectively. The array `pvt` encodes the permutation P in (6.1) as a sequence of interchanges. Specifically, the vector Px can be computed by the following fragment.

```
do i=1,npvt
   temp = x(i); x(i) = x(pvt(i)); x(pvt(i)) = temp
end do
```

For more see the Pivot suite.

The decomposition is computed by the generic subroutine `Ludpp` whose calling sequence is

```
call Ludpp(lu, A, info)
```

where

`type(RmatLudpp), intent(inout), target ::  lu`
: On return `lu` contains the LU decomposition of `A`.

`type(Rmat), intent(in) ::  A`
: The `Rmat` whose LU decomposition is to be computed.

`integer, intent(out), optional ::  info`
: If this optional argument is present, `Ludpp` returns the `info` parameter from the LAPACK routine `DGETRF`. The normal return is `info=0`. If `info>0`, the `info`th diagonal of `U` is zero.

6.3. The Cholesky decomposition

Given a symmetric positive definite matrix A of order n there is an upper triangular matrix R such that

$$A = R^{\mathrm{T}}R.$$

The matrix R is called the Cholesky factor of A.

The container for the decomposition is defined type `RmatChol` defined by

```
type RmatChol
   type(Rmat) :: R        ! The R-factor
   logical :: companion   ! True if the decomposition is
                          ! associated with a Rmat of interest
end type RmatChol
```

where `R` represents the Cholesky factor. The use of `companion` is explained in §6.1.

The Cholesky decomposition of a Rmat of tag SP is computed by the generic subroutine Chol, whose calling sequence is

```
call Chol(chola, A, info)
```

where

`type(RmatChol), intent(inout), target :: chola`
: On return `chola` contains the Cholesky decomposition of A.

`type(Rmat), intent(in) :: A`
: The `Rmat` whose Cholesky decomposition is to be computed.

`integer, optional, intent(out) :: info`
: If this optional argument is present, `Chol` returns the `info` parameter from the LAPACK routine `DPOTRF`. The normal return is `info=0`. If `info>0`, the leading submatrix of `A` of order `info` is indefinite.

6.4. The QR decomposition

Let A be an $m\times n$ matrix with $m \geq n$. Then there is an orthogonal Q such that

$$Q^{\mathrm{T}}A = \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{6.2}$$

where R is an $n\times n$ upper triangular matrix. We call (6.2) the QR decomposition of A.

If we partition

$$Q = (Q_1\ \ Q_2),$$

where Q_1 is $m\times n$, then we can write

$$A = Q_1 R. \tag{6.3}$$

This version of the decomposition is sometimes called the QR factorization. It cannot do as many things as the full decomposition, but it requires much less memory when $m \gg n$.

If $m < n$ then we can write the decomposition in the form

$$A = QR \tag{6.4}$$

where Q is orthogonal and R is an $m \times n$ upper triangular matrix.

The MATRAN module RmatQR_m provides the means of computing the three decompositions (6.2), (6.3), and (6.4). The container is RmatQR, which has the following definition.

```
type RmatQR
   type(Rmat) :: Q        ! The Q-factor
   type(Rmat) :: R        ! The R-factor
   logical :: companion   ! True if The decomposition is
                          ! associated with a Rmat of interest
end type RmatQR
```

The decomposition is computed by the generic subroutine QR, whose calling sequence is

```
call QR(qra, A, fullq, mywork)
```

where

`type(RmatQR), intent(out), target ::  qra`
: The QR decomposition of A.

`type(Rmat), intent(in) ::  A`
: The Rmat whose QR decomposition is to be computed.

`logical, intent(in), optional ::  fullq`
: If fullq is absent or present and false, QR computes the decomposition (6.3) or (6.4), depending on the row and column dimensions of A. If fullq is present and true, QR computes the decomposition (6.2) or (6.4), depending on the row and column dimensions of A

`real(wp), pointer, optional::  mywork(:)`
: The LAPACK subroutine DGEQRF requires an auxiliary work array, which is ordinarily allocated and deallocated by QR. If mywork is present and contains enough storage, it is used as the work array. If it is present but does not contain enough storage, it is reallocated and used as the work array. This storage is not deallocated, so that mywork can be reused when QR is called in a loop.

6.5. The pivoted QR decomposition

Let A be an $m \times n$ matrix with $m \geq n$. Then there is an orthogonal matrix Q and a permutation matrix P such that

$$Q^{\mathrm{T}}AP = \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{6.5}$$

where R is an $n \times n$ upper triangular matrix. The matrix P is formed by a process of column pivoting that results in a matrix R such that

$$r_{kk}^2 \geq \max_{j>k}\{\textstyle\sum_{i\geq k}|r_{ij}|^2\}.$$

This decomposition is called the pivoted QR decomposition or the QRP decomposition. If we partition

$$Q = (Q_1\ \ Q_2),$$

where Q_1 is $m \times n$, then we can write

$$AP = Q_1 R. \tag{6.6}$$

This version of the decomposition is sometimes called the pivoted QRP factorization of A.

If $m < n$ then we can write the decomposition in the form

$$AP = QR \tag{6.7}$$

where Q is orthogonal and R is an $m \times n$ upper triangular matrix.

The MATRAN module `RmatQRP_m` provides the means of computing the three decompositions (6.5), (6.6), and (6.7). The container is `RmatQRP`, which has the following definition.

```
type RmatQRP
   type(Rmat) :: Q               ! The Q-factor
   type(Rmat) :: R               ! The R-factor
   integer, pointer :: pvt(:)    ! The pivot array
   logical :: companion          ! True if The decomposition is
                                 ! associated with a Rmat
                                 ! of interest
end type RmatQRP
```

The array `pvt` encodes the permutation P in as a sequence of interchanges. Specifically the vector $x^{\mathrm{T}}P$ can be computed by the following fragment.

```
do i=1,A.m
   temp = x(i); x(i) = x(pvt(i)); x(pvt(i)) = temp
end do
```

The decomposition is computed by the generic subroutine `QRP`, whose calling sequence is

```
call QRP(qrpa, A, fullq, firstcols, mywork)
```

where

`type(RmatQR), intent(out), target :: QR`
The QR decomposition of `A`.

`type(Rmat), intent(in) :: A`
The `Rmat` whose QR decomposition is to be computed.

`logical, intent(in), optional :: fullq`
If `fullq` is absent or present and false, `QR` computes the decomposition (6.6) or (6.7), depending on the row and column dimensions of `A`. If `fullq` is present and true, `QR` computes the decomposition (6.5) or (6.7), depending on the row and column dimensions of `A`

`logical, intent(in), optional, target :: firstcols(:)`
If present, the columns $A(:,j)$ of A for which `firstcols(j)` is true are moved to the beginning of A and frozen there during the pivoting process. The length of `firstcols` may be less than `A%ncol`.

`real(wp), pointer, optional:: mywork(:)`
The LAPACK subroutine `DGEQRP` requires an auxiliary work array, which is ordinarily allocated and deallocated by `QRP`. If `mywork` is present and contains enough storage, it is used as the work array. If it is present but does not contain enough storage, it is reallocated and used as the work array. This storage is not deallocated, so that `mywork` can be reused when `QRP` is called in a loop.

6.6. The spectral decomposition

Let A be a symmetric matrix of order n. Then there is an orthogonal matrix V such that

$$A = VDV^{\mathrm{T}} \tag{6.8}$$

where $D = \mathrm{diag}(\delta_1, \ldots, \delta_n)$ with $\delta_1 \geq \cdots \geq \delta_n$. The scalars δ_i are the eigenvalues of A and the columns v_i of V are the corresponding eigenvectors. The decomposition (6.8) is called the spectral decomposition of A.

The MATRAN module `RmatSpec_m` defines and computes the type `RmatSpec`, which has the following definition.

```
type RmatSpec
   type(Rdiag) :: D         ! The matrix of eigenvalues.
   type(Rmat)  :: V         ! The matrix of eigenvectors.
   logical     :: companion ! True if the decomposition is
                            ! associated with a Rmat of interest
end type RmatSpec
```

The spectral decomposition is computed by the generic subroutine `Spec`, whose calling sequence is

```
call Spec(S, A, wantv, info, mywork)
```

where

`type(RmatSpec), intent(out) :: S`
: The spectral decomposition of `A`.

`type(Rmat), intent(in) :: A`
: The symmetric `Rmat` whose spectral decomposition is to be computed.

`logical, optional, intent(in) :: wantv`
: If `wantv` is present and true, compute both eigenvalues and eigenvectors. Otherwise compute only eigenvalues.

`integer, optional, intent(out) :: info`
: If present `info` returns the info parameter of the LAPACK routine `DSYEV`. The normal return is `info=0`. If `info>0`, `DSYEV` failed to converge.

`real(wp), pointer, optional :: mywork(:)`
: The LAPACK subroutine `DSYEV` requires an auxiliary work array, which is ordinarily allocated and deallocated by `Spec`. If `mywork` is present and contains enough storage, it is used as the work array. If it is present but does not contain enough storage, it is reallocated and used as the work array. This storage is not deallocated, so that `mywork` can be reused when Spec is called in a loop.

6.7. The singular value decomposition

Let A be an $m \times n$ matrix with $m \geq n$. Then there are orthogonal matrices U and V of order m and n such that

$$A = U \begin{pmatrix} D \\ 0 \end{pmatrix} V^{\mathrm{T}}, \tag{6.9}$$

where

$$D = \mathrm{diag}(\delta_1, \ldots, \delta_n)$$

with

$$\delta_1 \geq \cdots \geq \delta_n.$$

The decomposition (6.9) is called the singular value decomposition of A. The δ_i are called the singular values of A, and the columns of U and V are called the left and right singular vectors of A

If we partition $U = (U_1 \;\; U_2)$, where U_1 has n columns, then we may write

$$A = U_1 D V^{\mathrm{T}}. \tag{6.10}$$

The decomposition (6.10) is sometimes called the singular value factorization of A.

If $m < n$ the singular value decomposition assumes the form

$$A = U(D\ \ 0)V^{\mathrm{T}}, \tag{6.11}$$

where D is now of order m. Partitioning $V = (V_1\ \ V_2)$, where V_1 has m columns, we can write

$$A = UDV_1^{\mathrm{T}} \tag{6.12}$$

The module `RmatSdv_m` computes one of the decompositions (6.9), (6.10), (6.11), or (6.12). The decomposition is contained in the derived type `RmatSvd`.

```
type RmatSVD
      type(Rdiag) :: D          ! The singular values
      type(Rmat)  :: U          ! The right singular vectors
      type(Rmat)  :: V          ! The left singular vectors
      logical     :: companion  ! True if the decomposition is
                                ! associated with a Rmat
                                ! of interest
   end type RmatSVD
```

The decomposition is computed by the generic subroutine `SVD`, whose calling sequence is

```
call SVD(svdcmp, A, wantu, wantv, full, info, mywork)
```

where

`type(RmatSVD), intent(out), target :: svd`
: The singular value decomposition of `A`

`type(Rmat), intent(in) :: A`
: The `Rmat` whose singular value decomposition is to be computed.

`logical, optional, intent(in) :: wantu`
: If present and true compute compute the left singular vectors.

`logical, optional, intent(in) :: wantv`
: If present and true compute compute the right singular vectors.

`logical, intent(in), optional :: full`
: If present and true, compute the full complement of singular vectors requested by `wantu` or `wantv`. Otherwise compute the factorizations (6.10) or (6.12).

`integer, optional, intent(out) :: info`
: If present `info` returns the `info` parameter of the LAPACK routine `DGESVD`. The normal return is `info=0`. If `info>0`, `DGESVD` failed to converge.

`real(wp), pointer, optional::  mywork(:)`

The LAPACK subroutine `DGESVD` requires an auxiliary work array, which is ordinarily allocated and deallocated by `SVD`. If `mywork` is present and contains enough storage, it is used as the work array. If it is present but does not contain enough storage, it is reallocated and used as the work array. This storage is not deallocated, so that `mywork` can be reused when `SVD` is called in a loop.

6.8. The real Schur decomposition

Let A be of order n. Then there is an orthogonal matrix U such that

$$A = UTU^{\mathrm{T}},$$

where T is block upper triangular with 1×1 and 2×2 blocks on its diagonal. The 1×1 blocks are the real eigenvalues of A. The 2×2 blocks contain the complex eigenvalues of A. Such a decomposition is called a *real Schur decomposition* of A. The 2×2 blocks can be standardized to have the form

$$\begin{pmatrix} r & b \\ c & r \end{pmatrix},$$

where $bc < 0$. It is easily verified that the real part of the eigenvalues of this block is r while the imaginary parts are $\pm\sqrt{|b|}\,\sqrt{|c|}$.[8]

The MATRAN module `RealSchur_m` contains the wherewithal to compute a standardized real Schur decomposition of a `Rmat A`. The container is

```
type RmatRealSchur
   type(Rmat) :: T                 ! The block upper triangular matrix
                                   ! of the decomposition.
   type(Rmat) :: U                 ! The orthogonal matrix of the
                                   ! decomposition.
   complex(wp), pointer :: D(:)    ! D containes the eigenvalues of T
                                   ! in the order the appear on the
                                   ! diagonal of T.
   logical           :: companion ! True if the decomposition is
                                   ! associated with a Rmat of
                                   ! interest.
```

The real Schur decomposition is computed by the subroutine `RealSchur`, whose calling sequence is

[8]This formula is preferable to its mathematical equivalent $\pm\sqrt{|bc|}$, which is subject to exponent exceptions.

```
call Schur(S, A, wantu, info, mywork)
```

where

`type(RmatRealSchur), intent(out) :: S`
: The real Schur decomposition of `A`.

`type(Rmat), intent(in) :: A`
: The `Rmat` whose real Schur decomposition is to be computed.

`logical, optional, intent(in) :: wantu`
: If present and true, compute `U` and `T`. Otherwise compute only `T`.

`integer, optional, intent(out)`
: If present `info` returns the `info` parameter of the LAPACK routine `DGEES`. The normal return is `info=0`. If `info>0`, `DGEES` failed to converge.

`real(wp), pointer, optional:: mywork(:)`
: The LAPACK subroutine `DGEES` requires an auxiliary work array, which is ordinarily allocated and deallocated by `RealSchur`. If `mywork` is present and contains enough storage, it is used as the work array. If it is present but does not contain enough storage, it is reallocated and used as the work array. This storage is not deallocated, so that `mywork` can be reused when `RealSchur` is called in a loop.

The order in which eigenvalues appear on the diagonal of T cannot be predicted. Thus it may be necessary to reorder the blocks. The subroutine `ReorderSchur`. moves diagonal a block up or down the diagonal of T by pairwise exchanges. Its calling sequence is

```
ReorderSchur(S, i1, i2, info)
```

where

`type(RealSchur), intent(inout) :: S`
: The real Schur decomposition whose blocks are to be reordered. On return the blocks will be reordered as described below. The contents of `S%U` (if present) and `S%D` will be changed appropriately, so that `S` is still a standardized real Schur decomposition of the original matrix.

`integer, intent(inout) :: i1, i2`
: The block beginning in row `i1` is moved by pairwise exchanges of blocks to the row `i2`. If `S%D(i1)` is the second of a pair of complex eigenvalues, `i1` is decremented by 1. On return `i2` points to the first row of the block in its final position, which may differ from its original value by ± 1. The parameters `i1` and `i2` may take any values from `1` to `n`.

`integer, optional, intent(out)`
If present, the `info` parameter from the LAPACK routine `DTREXC` is returned. A nonzero value indicates an error.

Reordering is a numerical procedure, and it can alter the blocks of T. In particular, block containing two complex eigenvalues can split into two blocks containing real eigenvalues (mostly when the imaginary parts are very small). However, two real eigenvalues can never merge to form a complex block.

6.9. The eigendecomposition

Let A be a nondefective matrix. Then there is a (generally complex) matrix X such that

$$X^{-1}AX = D \equiv \mathrm{diag}(\delta_1, \ldots, \delta_n). \tag{6.13}$$

The numbers δ_i are called the eigenvalues of A and the columns x_i of X are the corresponding eigenvectors, which satisfy

$$Ax_i = \delta_i x_i.$$

If $Y^{\mathrm{H}} = X^{-1}$, then the columns y_i of Y satisfy

$$y_i^{\mathrm{H}}A = \delta_i y_i^{\mathrm{H}}.$$

The y_i are called the left eigenvectors of A.

The module `RmatEig_m` provides the means to compute the decomposition (6.13). The container is

```
type RmatEig
   complex(wp), pointer :: D(:)     ! The eigenvalues
   complex(wp), pointer :: X(:,:)   ! The right eigenvectors
   complex(wp), pointer :: Y(:,:)   ! The left eigenvectors
   logical :: companion             ! True if the decomposition
                                    ! is associated with a Rmat
                                    ! of interest
end type RmatEig
```

Note that this decomposition is different from the others — the results are not returned in matrix types. This is because at this point we have not defined a complex matrix type. Later a container `CmatEig` will remedy this deficiency. However, the type `RmatEig` may still be useful to those who do not want to bear the burden of incorporating the complex types into their programs.

The decomposition (6.13) is computed by the generic routine `Eig`, whose calling sequence is the following.

```
Eig(eiga, A, wantx, wanty, info, xwork, ywork, wwork)
```

where

`type(RmatEig), intent(out) :: eiga`
: The eigendecomposition of `A`

`type(Rmat), intent(in) :: A`
: The `Rmat` whose eigendecomposition is to be computed.

`logical, optional, intent(in) :: wantx`
: If present and true, compute right eigenvectors.

`logical, optional, intent(in) :: wanty`
: If present and true, compute left eigenvectors.

`integer, optional, intent(out) :: info`
: If present `info` returns the `info` parameter of the LAPACK routine `DGEEV`. The normal return is `info=0`. If `info>0`, `DGEEF` failed to converge.

`real(wp), pointer, optional :: rv(:,:), lv(:,:), mywork(:)`
: The LAPACK Routine `DGEEV` requires an auxiliary work arrays, which are ordinarily allocated and deallocated by `EIG`. If any of these three arrays is present present it is used, perhaps after a reallocation. This storage is not deallocated, so that the arrays can be reused when `EIG` is called in a loop.

7. The real core

At present MATRAN is a small package, and one can explicitly use only the modules one desires. As it grows, however, it will be desirable to define a core of modules that represents most of the needs of a typical program. The module in Figure 7.1 is an attempt at a beginning. What it leaves out is more significant than what it includes. The modules `RmatInv_m` and `RmatNorm2_m` are excluded because their use can be a source of unnecessary computation. All the major decompositions, excepting the LU and Cholesky decompositions, have been left out, on the grounds most programs need only a small selection of decompositions. The LU and Cholesky decomposition are included because they are used by `RmatSolve_m`.

Of course there is nothing to prevent the MATRAN user with special needs from defining a different list of modules. Only, please, do not call it `RealCore_m`.

8. Computing Arnoldi decompositions

In this section we give a more extended example of MATRAN's capabilities. Let A be a matrix of order n. An Arnoldi decomposition of A of order m is a relation of the form

$$AU_{m-1} = U_m B_{m,m-1}, \tag{8.1}$$

```
module MatranRealCore_m

   ! Root module

   use MatranUtil_m

   ! The two matrix objects

   use Rmat_m; use Rdiag_m

   ! Matrix operations

   use RmatTranspose_m; use RmatSum_m;  use RmatProduct_m
   use RmatSolve_m    : use RmatJoin_m; use RmatBorder_m
   use RmatSubmatrix_m

   use RdiagSum_m; use RdiagProduct_m; use RdiagSolve_m

   ! Matrix Miscelania

   use RdiagDiag_m; use RmatEye_m;   use RmatNorm_m;
   use RmatPivot_m; use RmatPrint_m; use RmatRand_m

   ! Decompositions

   use RmatLudpp_m; use RmatChol_m

end module RealCore_m
```

Figure 7.1: The MATRAN real core

where U_m is an orthonormal matrix with m columns, U_{m-1} consists of the first $m-1$ columns of U_m and B is an $m\times(m-1)$ upper Hessenberg matrix. As the order of an Arnoldi decomposition increases, the matrices $B_{m-1,m-1}$, consisting of the first $m-1$ rows of $B_{m,m-1}$ generally contain increasingly accurate approximations to the extreme eigenvalues of A. Approximate eigenvectors can also be extracted from U_{m-1}, by a process known as the Rayleigh–Ritz method.

If we denote by U_k the matrix consisting of the first k columns of U_m and $B_{k,k-1}$

the leading $(k+1)\times k$ submatrix of $B_{m,m-1}$, then

$$AU_{k-1} = U_k B_{k,k-1} \tag{8.2}$$

is also an Arnoldi decomposition of A. This suggests that we compute (8.1) by forming a sequence of Arnoldi decompositions each computed from the previous one. Here is the algorithm for passing from the decomposition (8.2) to the next.

$$\begin{array}{ll} 1. & u_{k+1} = Au_k \\ 2. & r = U_k^{\mathrm{T}} u_{k+1} \\ 3. & u_{k+1} = u_{k+1} - U_k r \\ 4. & \rho = \|u_{k+1}\|_2 \\ 5. & u_{k+1} = u_{k+1}/\rho \\ 6. & U_{k+1} = (U_k \ \ u_{k+1}) \\ 7. & B_{k+1,k} = \begin{pmatrix} B_{k,k-1} & r \\ 0 & \rho \end{pmatrix} \end{array} \tag{8.3}$$

The process must be started with a vector u_1. In our example u_1 will be a normalized random vector.

Steps 3–5 in this algorithm orthogonalize Au_k against U_k and normalize it, a process known as Gram–Schmidt orthogonalization. Unfortunately, the process can fail, and we use a more complicated process called Gram–Schmidt with reorthogonalization.

The following code shows implements the Arnoldi process. It consists of a main program `Arnoldi` and three subroutines:

`ArnStep`
: Implements the algorithm (8.3).

`gsro`
: Performs Gram–Schmidt with reorthogonalization.

`Amult`
: Multiplies a vector by A. In this case $A = \mathrm{diag}(1, 0.95, 0.95^2, \ldots, 0.95^{n-1})$.

For convenience these routines are made local to the program `Arnoldi`.

```
program Arnoldi

use Rmat_m
use RmatSum_m
use RmatProduct_m
use RmatNorm_m
use RmatRand_m
use RmatSolve_m
```

```
use RmatBorder_m
use RmatPrint_m
use RmatEye_m
use RmatEig_m
use RmatSubmatrix_m

implicit none

!  Let U_m = (u_1,...,u_m) be orthonormal and let B_{m,m-1}
!  be an mx(m-1) upper Hessenberg matrix.  If
!
!   (*)   AU_{m-1} = U_mB_{m,m-1},
!
!  then (*) is called an Arnoldi decomposition of A.  An Arnoldi
!  decomposition can be built up sequentially by starting with a
!  normalized vector u_1.  Given U_{k-1}, u_{k} is generated by
!  orthonormalizing Au_{k-1} against the columns of U_{k-1}.  The
!  orthogonalizing coefficients form the k-th column of B_{k,k-1}.
!  The eigevalues of B_{k-1,k-1} often contain increasingly accurate
!  approximations to the extreme eigenvalue of A.
!
!   This program compute an Arnoldi decomposition starting from a
!  normalized random vector.  It also computes the dominant eigenvalue
!  of B_{k-1,k-1} to show its convergence.  It uses the subroutine
!  ArnStep to advance the decomposition.  ArnStep in turn uses Amult
!  to multiply a vector by the matrix in question and gsro
!  (Gram-Schmidt with reorthogonalization) to perform the
!  reorthogonalization.

   type(Rmat) :: U, B
   type(RmatEig) :: eigb

   integer :: bigeigloc(1), k, n, m

   ! Get the order n of A and the number of
   ! Arnoldi vectors to compute.

   print *, 'Input n and m'
   read *, n, m

   ! Initialize storage for U and B

   U = (/n,0, n,m/)
   B = (/0,0, m+1,m/)
```

```
   ! Compute the Arnoldi decomposition.

   call random_seed() ! Initialize the random number generator.

   do k=0,m-1

      ! Advance the decomposition.

      call ArnStep(U, B)

      ! Compute and print the largest eigenvalue of
      ! the Rayleigh quotient B(1:k,1:k)

      if (k>0) then
         call Eig(eigb, Sbm(B, 1,k, 1,k))
         bigeigloc = maxloc(abs(eigb%D(1:k)))
         print '(e23.15,  e9.1)', eigb%D(bigeigloc(1))
      end if
   end do

   ! Check the defining relations of the final
   ! Arnoldi decomposition.

   print *, ' '
   print *, NormF(.xhx.U - Deye(m)), &
            NormF(Amult(Col(U, 1,m-1)) - U*B)

contains

   subroutine ArnStep(U, B)
      type(Rmat), intent(inout) :: U, B

      ! ArnStep takes expands an Arnoldi decomposition ! of order k to
      ! one of order k+1.  If k=0, ArnStep ! initializes the
      ! decomposition to a random vector.

      type(Rmat) :: x, xp, r
      real(wp) rho
      integer k

      n = U%nrow
      k = U%ncol

      ! Get a starting vector for the Krylov sequence.
```

```
      if (k==0) then
         U = DRandN(n,1)
         U = U/NormF(U)
         call ReshapeAry(B, 1, 0)
         return
      end if

      ! Compute Au_k, orthogonalize it, and fold the results
      ! into U and B.

      x = Amult(col(U,k))

      call gsro(U, x, xp, r, rho)

      call BorderE(U, xp)
      call BorderSE(B, .dm.(/1,k-1/), r, .dm.rho)

   end subroutine ArnStep

   subroutine gsro(Q, x, xp, r, rho)

      type(Rmat), intent(in) :: Q, x
      type(Rmat), intent(out) :: xp, r
      real(wp), intent(out) :: rho

!     gsro orthogonalizes a column vector x against the the columns of
!     the orthonormal matrix Q to produce a normalized vector xp that
!     is orthogonal to Q to working accuracy.  Moreover, the relation
!
!        x = Q*r + rho*xp
!
!      is satisfied to working accuracy.  The method used is
!      Gram-Schmidt with reorthogonalization.

      real(wp), parameter :: run = 2.2d-16 ! Rounding unit.

      real(wp) :: nu, sig, tau
      type(Rmat) :: s, xp

      call GuardTemp(Q)
      call GuardTemp(x)

      nu = NormF(x)
      r = .dm.(/Q%ncol,1/)
```

```
      !Special action for null Q

      if (Q%ncol == 0) then
         xp = x/nu
         rho = nu
         go to 99999
      end if

      sig = nu
      xp = x
      do

         ! Orthogonalize.

         s = Q.xhy.xp
         r = r + s
         xp = xp - Q*s
         tau = NormF(xp)

         ! Finished if reduction in norm is not too great.

         if (tau > 0.5*sig) exit

         ! If the current norm of xp has not dropped
         ! below the 0.1 times the rounding unit relative
         ! to original norm of xp, continue orthogonalizing.
         ! Otherwise replace xp by a small random vector.

         if (tau > 0.1*nu*run) then
            sig = tau
         else
            nu = 0.1*nu*run
            sig = nu
            call RandN(xp, xp%nrow, 1)
            xp = sig*(xp/normf(xp))
         end if
      end do

      ! Normalize and return.

      rho = NormF(xp)
      xp = xp/rho

99999&
      call CleanTemp(Q)
```

```
      call CleanTemp(x)

   end subroutine gsro

   function Amult(x) result(y)
      type(dmat) :: y
      type(dmat), intent(in) :: x

      ! Amult computes the product y = Ax, where
      ! A = diag(1, .95, .95^2, ..., .95^{n-1}).

      integer :: i
      real(wp) :: s

      call GuardTemp(x)

      y%a => null() ! Necessary because the SUN f95 6.2
      call Clean(y) ! does not initialize the results of
                    ! functions properly.

      y = x
      s = 1.0
      do i=1,y%nrow
         y%a(i,1:y%ncol) = s*y%a(i,1:y%ncol)
         s = 0.95*s
      end do

      call CleanTemp(x)

   end function Amult

end program Arnoldi
```

9. Appendix: The Sun Fortran 95 6.2 Compiler

When the result of a function is a defined type, the Sun Fortran 95 6.2 Compiler may not initialize it properly. The following code (implementing an aspect of .dm.) shows the necessary fix.

```
! RmFromAry overloads .dm. to produce C = ary.

function RmFromAry(ary) result(C)
   type(Rmat) :: C
   real(wp), intent(in) :: ary(:,:)


   C%a => null()          ! Nullify the C%a and C%temporary
   C%temporary => null()  ! and call Clean to initialize
   call Clean(C)          ! the other components.

   C = ary
   call SetTemp(C)

end function RmFromAry
```

Since I developed MATRAN on a Sun system, all code has been thoroughly sun-screened. The fix will be removed as soon as Sun fixes the problem.