Interfaces and derived types in Fortran

Abstract

In this assignment, the focus is on the creation of a derived data type for the handling of complex matrices. This new data type must contain the dimensions of the matrix, the elements of the matrix, the trace and the determinant. Furthermore, functions dedicated to the calculation of the adjoint and of the trace of the matrix are developed. By using the `INTERFACE` functionality, such functions are defined for both plain matrices and the new data type. Finally, by defining a dedicated subroutine, the contents of an instantiation of the new data type are written on an external, human-readable file.

Theory

The *adjoint* of a matrix A, which is here denoted $A^\dagger$, was understood to be the *conjugate transpose* of A, thus being an *hermitian adjoint*. The trace of the adjoint can be easily calculated as follows:

$$\mathrm{Tr}(A^\dagger) = \overline{\mathrm{Tr}(A)}, \tag{1}$$

while the following can be used in order to compute its determinant:

$$\det(A^\dagger) = \overline{\det(A)}. \tag{2}$$

Code development

As the assignment does not explicitly mention whether the matrices we are dealing with are square, the most general choice is to allow rectangular matrices. This, however, poses a problem when calculating the trace, the determinant and the (hermitian) adjoint, as these quantities are only properly defined for square matrices.

In the case of non-square matrices, the following policies were adopted:

- **Trace**: the trace is set to 0, and an error message is printed on screen;
- **Adjoint**: there is no computational reason to forbid the transposition and the conjugation of a rectangular matrix, so the operation is carried out; however, a warning is issued to the user.

The implementation of the derived data type, called `RICHMAT` (as it effectively is an 'enriched matrix'), and its related functions, proceeded as follows. In the first place, the data type itself was deployed.

```
TYPE RICHMAT
   ! This dtype is an "enriched matrix":
   ! - dims: a vector of integers of size 2
   ! - elems: a double precision complex matrix; its dimensions should
   !   be those stored in dims
   ! - trace: a double precision complex number, representing the
```

```
    !   trace of the matrix elems
    ! - det: a double precision complex number, representing the
    !   determinant of the matrix elems
    ! TO BE ADDED:
    ! - updated : a logical flag to denote if the RICHMAT was updated
    INTEGER*4, DIMENSION(2) :: dims
    COMPLEX*16, DIMENSION(:,:), ALLOCATABLE :: elems
    COMPLEX*16 :: trace, det
    !LOGICAL :: updated
  END type RICHMAT
```

Note how the matrix inside `RICHMAT` is set to be `ALLOCATABLE`, allowing for arbitrary dimensions. An `updated` flag was also predisposed for future use; in the case of a change in `elems` without also changing `trace` and `det` accordingly, the `updated` flag would be set, for example, to `.FALSE.`, notifying the discrepancy.

An initializer was then created, taking two integer values (which are then used as the dimensions of the matrix inside `RICHMAT`) as input, setting everything to 0. If either of the two dimensions are less or equal than 0, a warning is issued, and the matrix is set to (0,0) dimension.

```
  FUNCTION NULL_INIT(mat_size1, mat_size2)
    ! given the size, this initialization creates a null RICHMAT. If either
    ! one of the dimensions is null or negative, the matrix is left undefined,
    ! having both dimensions set to 0. An error message is also displayed.
    ! - mat_size1, mat_size2 : integers containing the number of rows and
    !   cols of the matrix, respectively
    INTEGER*4, INTENT(IN) :: mat_size1, mat_size2
    TYPE(RICHMAT) :: NULL_INIT
    if ((mat_size1 .ge. 1) .and. (mat_size2 .ge. 1)) then
       ALLOCATE(NULL_INIT%elems(mat_size1,mat_size2))
       NULL_INIT%dims = (/ mat_size1, mat_size2 /)
       NULL_INIT%elems = 0.d0
       NULL_INIT%trace = 0.d0
       NULL_INIT%det = 0.d0
    else
       print*, "Error: the structure could not be properly created: &
&negative dimension."
       ALLOCATE(NULL_INIT%elems(0,0))
       NULL_INIT%dims = (/ 0, 0 /)
       NULL_INIT%elems = 0.d0
       NULL_INIT%trace = 0.d0
       NULL_INIT%det = 0.d0
    endif
    RETURN
  END FUNCTION NULL_INIT
```

The functions `trace` and `adjoint` were initially implemented for use with a `RICHMAT` data type. The whole `RICHMAT` is passed; the result of `trace` is simply to return a `COMPLEX*16`, while `adjoint` returns a `RICHMAT` with all its values properly calculated using the relations in the Theory section.

```fortran
FUNCTION TRACE_RICH(rich_mat)
  ! this function is rather useless; it simply returns the trace
  ! of the RICHMAT given in input. It is effectively equivalent to
  ! writing RICHMAT%trace.
  ! - rich_mat : a RICHMAT
  TYPE(RICHMAT), INTENT(IN) :: rich_mat
  COMPLEX*16 :: trace_rich
  INTEGER*4 :: ii
  if (rich_mat%dims(1) .eq. rich_mat%dims(2)) then
     trace_rich = 0.d0
     do ii=1, rich_mat%dims(1)
        trace_rich = trace_rich + rich_mat%elems(ii,ii)
     enddo
  else
     print*, "Matrix is not square: trace set to 0."
     trace_rich = 0.d0
  endif
  RETURN
END FUNCTION TRACE_RICH
```

```fortran
FUNCTION RICH_ADJOINT(rich_mat)
  ! given a RICHMAT, this function computes the adjoint of the
  ! elements of the input, and bundles it together with the other
  ! components. From a mathematical point of view, the adjoint is only
  ! defined for square matrices. However, no computational reason to
  ! not allow for its calculation. As such, no check on the dimensions
  ! is performed.
  ! - enriched_mat : a RICHMAT, the info of which are used to create
  !   its adjoint
  TYPE(RICHMAT), INTENT(IN) :: rich_mat
  TYPE(RICHMAT) :: rich_adjoint
  if (rich_mat%dims(1) .ne. rich_mat%dims(1)) print*, "WARNING: non-square matrix."
  rich_adjoint%trace = conjg(rich_mat%trace)
  rich_adjoint%det = conjg(rich_mat%det)
  rich_adjoint%dims = rich_mat%dims(2:1:-1)
  rich_adjoint%elems = transpose(conjg(rich_mat%elems))
  RETURN
END FUNCTION RICH_ADJOINT
```

In order to make the `INTERFACE` block actually useful, the `trace` and `adjoint` functions were deployed also for plain matrices. Of particular interest is the fact that the plain matrix versions were written without taking the size of the matrix as input. This resulted in both plain and rich functions to have *just one* input; since we interfaced the functions as *operators*, this resulted in no ambiguity in the way such operator worked. In other words, the syntaxes of both the `trace` and the `adjoint` operators are the same, regardless of them acting on plain or rich matrices.

This was prompted by the errors that arose when trying to call `trace` on a plain matrix: it required the size as input, but `.trace.(matsize,matelems)` resulted in an error, because the `trace` operator required the following syntax: `matsize.trace.matelems`.

An example of the size-less `trace` function follows.

```fortran
FUNCTION TRACE_MAT(mat_elems)
  ! given a matrix (equivalently, the elements of a square RICHMAT),
  ! this function checks whether the matrix is square and then, if it
  ! is, calculates the trace.
  ! - mat_size : an integer 2-vector, representing the number of cols/
  !   rows of mat_elems
  ! - mat_elems : a double precision complex matrix containing the
  !   elements
  ! - trace_mat : a double complex number, containing the trace of the
  !   matrix
  INTEGER*4, DIMENSION(2) :: mat_size
  INTEGER*4 :: ii
  COMPLEX*16, DIMENSION(:,:), INTENT(IN) :: mat_elems
  COMPLEX*16 :: trace_mat
  mat_size=SHAPE(mat_elems)
  if (mat_size(1) .eq. mat_size(2)) then
     trace_mat = 0.d0
     do ii=1, mat_size(1)
        trace_mat = trace_mat + mat_elems(ii,ii)
     enddo
  else
     print*, "Matrix is not square: trace set to 0."
     trace_mat = 0.d0
  endif
  RETURN
END FUNCTION TRACE_MAT
```

Finally, a subroutine dedicated to writing a `RICHMAT` on file was implemented. The most critical part was the writing of the matrix elements. A simple `do` cycle was used, together with slicing, to print the matrix one row at a time. This was preferred to a more elaborate writing function (which, for instance, could write the numbers in the form $a + bi$), because later access to the data may be more difficult. In the tradeoff between readability and usability, the latter was preferred.

Finally, the plain matrices and the matrices inside the `RICHMAT`s were deallocated.

Results

The test file produces a random `RICHMAT`, then its adjoint, and prints everything to two separate text files. It also uses the `.trace.` operator on two plain matrices, one the adjoint of the other, with no error. After various tests, no problems have arisen. An example of the output file follows.

```
Dimensions:             3            3

Trace:                (1.2347366735339165,0.70889285951852798)

Determinant:                  (0.0000000000000000,0.0000000000000000)

Matrix elements:
```

```
(0.11377593129873276,3.46478149294853210E-002)
 ↪  (0.25841724872589111,0.47480273246765137)
 ↪  (0.63709616661071777,0.49770611524581909)
     (0.12620790302753448,0.98060971498489380)
      ↪  (0.65527600049972534,0.12371765077114105)
      ↪  (0.36357876658439636,0.22294211387634277)
     (0.16453519463539124,0.24889101088047028)
      ↪  (0.10088932514190674,0.79697477817535400)
      ↪  (0.46568474173545837,0.55052739381790161)
```

Comments and self-evaluation

This assignment was instrumental for me to learn how structures can be implemented in Fortran. I realized the difference between operators and functions, and, in the process, I also learned how to write functions on arrays/matrices without having to pass their size as input.

The next step may be to include some degree of security in the code, in order to avoid direct access to a derived data type's components. To do so, `Public` and `Private` access may be implemented. On the same note, an additional boolean variable `updated` may be included in the `RICHMAT` data type, as to keep track of discrepancies in the values inside it. Finally, a 'destructor' subroutine could be used, in order to deallocate those `RICHMAT`s which are no more useful.