# Fortran derived types

**Abstract**

The aim of this exercise is to write a `MODULE` which contains a double complex matrix derived `TYPE` that will be use to handle complex matrices. Moreover, we will define some functions/subroutines that will compute the trace and the adjoint of the matrix. Finally, we will define a subroutine that writes on a file the matrix `TYPE` in a readable form.

## 1 Theory

The trace of an $n \times n$ square matrix $A$ is defined as

$$\mathrm{Tr}(A) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \cdots + a_{nn}, \tag{1}$$

where $a_{ii}$ denotes the entry on the $i$-th row and $i$-th column of $A$. It is the sum of its (complex) eigenvalues, and it is invariant with respect to a change of basis.
The adjoint of $A$ (with dense domain $D(A)$), denoted as $A^\dagger$, is defined as

$$(A^\dagger \phi, \psi) = (\phi, A\psi), \qquad \forall \psi \in D(A), \phi \in D(A^\dagger) \tag{2}$$

with domain

$$D(A^\dagger) = \left\{ \phi \in \mathcal{H} \, s.t. \sup_{\psi \in D(A)} \frac{|(\phi, A\psi)|}{\|\psi\|} < \infty \right\} \tag{3}$$

where $\mathcal{H}$ is a Hilbert space. If $\dim(\mathcal{H}) < \infty$, then $A^\dagger$ is simply the conjugate transpose of $A$:

$$(A\psi, \phi) = \sum_{i,j=1}^{N} (A_{ij}\psi_i)^* \phi_j = (\psi, A^\dagger \phi) = \sum_{i,j=1}^{N} \psi_i^* A_{ij}^* \phi_j \qquad \Rightarrow \qquad A^\dagger_{ij} = A^*_{ji}.$$

The trace and the determinant of the adjoint matrix can be computed as follows:

$$\mathrm{Tr}\big(A^\dagger\big) = \overline{\mathrm{Tr}(A)}, \qquad \det\big(A^\dagger\big) = \overline{\det(A)}. \tag{4}$$

## 2 Code development

First, we write a `module matrices` which contains the `type dmatrix` we want to define.

```fortran
    type dmatrix
        integer, dimension(2) :: dim
        double complex, dimension(:, :), allocatable :: elem
        double complex :: tr, det
    end type
```

This `type` includes four components:

- `dim`, a vector which contains the dimensions of the matrix;
- `elem`, the matrix itself;
- `tr`, the trace of the matrix;

- `det`, the determinant of the matrix.

Then we have to define the interfaces for the functions we are going to define afterwards.

```
    interface operator (.init.)
        module procedure MatInit
    end interface operator (.init.)

    interface operator (.trace.)
        module procedure MatTrace
    end interface operator (.trace.)

    interface operator (.adj.)
        module procedure MatAdj
    end interface operator (.adj.)
```

Now, after the `contains` statement we can define the functions we have specified in the interfaces. First we have the function `MatInit`, which takes as input a vector containing the dimensions of the matrix and gives as output a `dmatrix` type matrix which elements are all set to zero. Additionally, it sets also the trace and the determinant to zero.

```
    function MatInit(dimensions) result(M)
        integer, dimension(2), intent(in) :: dimensions
        type(dmatrix) :: M
        if ((dimensions(1) .le. 1) .and. (dimensions(2) .le. 1)) then
            print *, "[MatInit warning] Non valid matrix dimension."
        else
            allocate(M%elem(dimensions(1), dimensions(2)))
            M%dim = dimensions
            M%elem = 0.d0
            M%tr = 0.d0
            M%det = 0.d0
        end if
        return
    end function MatInit
```

Then we have the function `MatTrace`, which computes the trace of the matrix that is given as input only if it is square. If it is not, the trace is set to zero.

```
    function MatTrace(M) result(trace)
        type(dmatrix), intent(in) :: M
        double complex :: trace
        integer :: ii
        trace = 0.d0
        if (M%dim(1) .ne. M%dim(2)) then
            print *, "[MatTrace warning] The matrix is not square. The trace will
    be set to zero."
            trace = 0.d0
        else
            do ii = 1, M%dim(1)
                trace = trace + M%elem(ii, ii)
            end do
        end if
        return
    end function MatTrace
```

The last function, `MatAdj`, computes the adjoint matrix of the one given as input, together with its components `tr`, `det` and `dim`, according to the theory.

```
    function MatAdj(M) result(Madj)
        type(dmatrix), intent(in) :: M
        type(dmatrix) :: Madj
        Madj%tr = conjg(M%tr)
        Madj%det = conjg(M%det)
        Madj%dim = M%dim(2:1:-1)
        Madj%elem = transpose(conjg(M%elem))
        return
    end function MatAdj
```

Finally, we define a `subroutine` that writes the full matrix, the dimension, the trace and the determinant in a text file.

```
    subroutine WriteResults(M, filename)
        type(dmatrix) :: M
        character*8 :: filename
        integer :: ll
        open(unit = 2, file = filename, action = "write", status = "replace")
        write(2, *) "MATRIX:"
        do ll = 1, M%dim(1)
            write(2, *) M%elem(ll, :)
        end do
        write(2, *) " "
        write(2, *) "DIMENSION: rows =",  M%dim(1), ",  columns = ", M%dim(2)
        write(2, *) " "
        write(2, *) "TRACE:", M%tr
        write(2, *) " "
        write(2, *) "DETERMINANT:", M%det
        close(2)
        return
    end subroutine WriteResults
```

Notice that the matrix elements, which are of the form $a + ib$, are written using the notation $(a, b)$ for simplicity and usability.

The idea behind the program is to ask the user to enter the dimension of the matrix and then execute all the computations independently. So, when executed, the program will prompt the following line:

```
    Please enter the dimension of the matrix [nrows, ncols]:
```

If the number of rows or columns is less or equal than one, the user will be notified and asked to insert again the dimension of the matrix.

```
    Non valid dimension! The number of rows and columns must be greater than 1.
    Please enter the dimension of the matrix [nrows, ncols]:
```

Since the trace is defined only for square matrices, if the user does not provide a number of rows which is equal to the number of columns he/she will be notified and asked if he/she wants to continue.

```
    Warining! The matrix is not square, so the trace is not defined and will be
    set to zero.
    Do you want to continue? [y/n]
```

If the user decides to continue (`y`) the trace will be set to zero, otherwise (`n`) he/she will be asked to insert the dimension of a *square* matrix.

Once inserted the the dimension of the matrix, the program will initialize it, fill it with random numbers between 0 and 1 (both for real and imaginary part) and compute the trace and the adjoint using the interface operators defined above.

```
    A = .init.(d)
    A%elem = cmplx(RealPart, ImPart)
    A%tr = .trace.(A)
    Aadj = .adj.(A)
```

## 3 Results

When executed, the program will produce two text files:

- `M.txt`, which contains the original matrix together with its dimension, trace and determinant;
- `Madj.txt`, which contains the adjoint matrix, together with its dimension, trace and determinant.

An example of `M.txt` is shown here:

```
MATRIX:
(0.51949119567871094,0.61248314380645752)   (0.35190868377685547,0.28647136688232422)
(0.28978204727172852,0.49185514450073242)   (0.30201876163482666,0.18747985363006592)

DIMENSION: rows =           2 ,  columns =           2

TRACE:            (0.82150995731353760,0.79996299743652344)

DETERMINANT:              (0.0000000000000000,0.0000000000000000)
```

The content of the corrispondent `Madj.txt` will be the following:

```
MATRIX:
(0.51949119567871094,-0.61248314380645752)    (0.28978204727172852,-0.49185514450073242)
(0.35190868377685547,-0.28647136688232422)    (0.30201876163482666,-0.18747985363006592)

DIMENSION: rows =           2 ,  columns =           2

TRACE:           (0.82150995731353760,-0.79996299743652344)

DETERMINANT:             (0.0000000000000000,-0.0000000000000000)
```

## 4 Self evaluation

This exercise was very instructive since it made me understand how derived types and interface operators work. Moreover, I learned that when declaring variables inside functions and subroutines that need to be passed in or out, `intent(in)` or `intent(out)` may be added to the declaration.

Further development of the code could include the implementation of the computation of the determinant. However, this seems to be a tough task, since even the LAPACK library does not provide a routine for computing determinants due to stability problems.