Università degli Studi di Padova

Dipartimento di Fisica e Astronomia “Galileo Galilei”

Corso di Laurea in Physics of Data

QUANTUM INFORMATION AND COMPUTATION: HOMEWORK 2

OF

TOMMASO TABARELLI

Anno accademico 2019-2020

Abstract

In this homework we are asked to create a new data structure, some operators and funtcions to ease matrix handling. In particular, the structure contained matrix elements (that can be complex), dimensions, trace, determinant. The operators created had the following tasks: evaluating the trace, calculate the adjoint of a matrix; to use them, the creation of interfaces was also needed. The required functions was meant to initialize matrix elements and print information about the matrix to a file in a human understandable way. The evaluation of the determinant was not asked in this homework.

Theory

A matrix *trace* is defined (for square matrix) as the sum of the elements on the main diagonal (from the upper-left to the lower-right). Formally, given a square matrix $A \in M(n,n)$, its trace is:

$$tr(A) = \sum_{i=0}^{n} a_{ii}$$

The *adjoint* of a matrix A is the transposed and conjugate of A itself. Formally:

$$A^* = \overline{A^T}$$

Code development

First thing to do was to define a module where to put all the stuff that will be defined: in this way all the code can be retrieved in future exercises; that module's name is **MATRICES**.

```
MODULE MATRICES
```

Next, a new type has been decleared: **DMATRIX**. It would include: matrix elements, dimensions, trace and determinant as aforementioned.

```
TYPE DMATRIX
    ! Elem  : matrix elements
    ! Dims  : matrix dimensions (Elem effective dimension and Dims should be equal)
    ! Trace : matrix trace
    ! Det       : matrix determinant
    COMPLEX*8, DIMENSION(:,:), ALLOCATABLE :: Elem
    INTEGER, DIMENSION(2) :: Dims
    COMPLEX*8 :: Trace, Det
END TYPE DMATRIX
```

Next step was to define the operators and the corresponding functions to use for operators calls. For operators the return type had to be chosen: since some operators (such as '+') return the same type as the operands while others no (such as the *vector inner product*).
In the code the choice was to implement both cases for the trace operator (even if the syntax does not allow to have functions with same arguments when dealing with operators, a trick was found to bypass this). In the first case, the argument is of type *DMATRIX* and a new one, enriched with the trace information, is returned; in the other case the argument is a 2 dimensional array (not a complete

DMATRIX) and the return is a *COMPLEX*8* number, representing the trace.
For trace, one had to decide how to handle the non-square matrices since the trace is defined only for square matrices. In the code, this choice was developed returning a "NULL" result; in other words, a DMATRIX structure was returned, having dimension (1,1) and all quantities set to $0+0i$.

```fortran
INTERFACE OPERATOR(.Trc.)
   MODULE PROCEDURE Mat_trc,Trace_num
END INTERFACE
```

```fortran
! TRACE FUNCTION returning type: DMATRIX
   FUNCTION Mat_trc(Matrix_)
   ! Matrix_  : TYPE DMATRIX

   IMPLICIT NONE

   TYPE(DMATRIX), INTENT(IN) :: Matrix_
   TYPE(DMATRIX) :: Mat_trc
   COMPLEX*8 :: Trc
   INTEGER :: ii

   ! Verifiyng the matrix is square and returning a "NULL" result if not
   IF (Matrix_%Dims(1).NE.(Matrix_%Dims(2))) THEN
      WRITE(*,*) "The matrix is not square. Returning a NULL result."
      Mat_trc%Dims = (0,0)
      ALLOCATE( Mat_trc%Elem(0,0) )
      Mat_trc%Elem = (0d0,0d0)
      Mat_trc%Trace = (0d0,0d0)
      Mat_trc%Det = (0d0,0d0)
      RETURN
   END IF

   Mat_trc%Dims(1) = Matrix_%Dims(1)
   Mat_trc%Dims(2) = Matrix_%Dims(2)

   ALLOCATE( Mat_trc%Elem(Mat_trc%Dims(1),Mat_trc%Dims(2)) )

   ! Initialize matrix (or suppose it is already initialized)
   Mat_trc%Elem = Matrix_%Elem

   Trc = 0d0

   Mat_trc%Trace = 0d0

   ! Evaluating trace (supposing the matrix is square)
   DO ii=1,Mat_trc%Dims(1)
      Trc = Trc + Matrix_%Elem(ii,ii)
   END DO

   Mat_trc%Trace = Trc
```

```
    RETURN
END FUNCTION Mat_trc
```

```
! TRACE FUNCTION returning type: COMPLEX*8
FUNCTION Trace_num(Matrix_elem)
    ! Matrix_elem  : TYPE COMPLEX*8 bidimensional array

    IMPLICIT NONE

    !TYPE(DMATRIX), INTENT(IN) :: Matrix_
    COMPLEX*8, DIMENSION(:,:), INTENT(IN) :: Matrix_elem
    COMPLEX*8 :: Trace_num
    INTEGER :: ii
    INTEGER*8, DIMENSION(2) :: dims

    Trace_num = 0d0

    ! Getting proper dimensions
    dims = SHAPE(Matrix_elem)

    ! Verifiyng the matrix is square and returning a "NULL" result if not
    IF (dims(1).NE.(dims(2))) THEN
        WRITE(*,*) "The matrix is not square. Returning a NULL result."
        Trace_num = (0d0,0d0)
        RETURN
    END IF

    ! Evaluating trace (supposing the matrix is square, so
    !   looping only in 1 dimension)
    DO ii=1,dims(1)
    Trace_num = Trace_num + Matrix_elem(ii,ii)
    END DO

    RETURN
END FUNCTION Trace_num
```

For the *adjoint* operator the choice was to implement it taking as argument a *DMATRIX* type and returning the same type as result; elements are transposed and conjugated, trace and determinant are conjugated. This operator had not issues dealing with non-square matrices.

```
INTERFACE OPERATOR(.Adj.)
    MODULE PROCEDURE Mat_adj
END INTERFACE
```

```
! ADJOINT FUNCTION
FUNCTION Mat_adj(Matrix_)
    ! Matrix_ is of TYPE DMATRIX : it is the input object

    IMPLICIT NONE
    INTEGER :: ii
    TYPE(DMATRIX), INTENT(IN) :: Matrix_
    TYPE(DMATRIX) :: Mat_adj

    Mat_adj%Dims(1)=Matrix_%Dims(2)
    Mat_adj%Dims(2)=Matrix_%Dims(1)

    Mat_adj%Trace = CONJG(Matrix_%Trace)

    Mat_adj%Det = CONJG(Matrix_%Det)

    Mat_adj%Elem = CONJG(TRANSPOSE(Matrix_%Elem))

    RETURN
END FUNCTION Mat_adj
```

The function to initialize the elements of the *DMATRIX* was designed to take as input the dimensions of the matrix and a complex value used to set all matrix elements. Trace and determinant are set to 0, expecting they are updated during the execution of a program.
A check on dimensions is performed, testing if they are positive values or not.

```
! Defining interface to use to initialize DMATRIX objects
INTERFACE init
    MODULE PROCEDURE Init_matr
END INTERFACE
```

```
! INITIALIZING FUNCTION
FUNCTION Init_matr(Dims, Value_)
    ! Dims  : matrix dimensions
    ! Value_   : value to initialize ALL matrix elements

    INTEGER, DIMENSION(2) :: Dims
    COMPLEX*8 :: Value_
    TYPE(DMATRIX) Init_matr
    INTEGER :: ii,jj

    Init_matr%Dims(1) = Dims(1)
    Init_matr%Dims(2) = Dims(2)

    ! Here it should be better to use a WHILE (I cant use it for now)
    !   to eventually ask again for dimensions, or eventually
    !   return a "NULL" result to be collected by a loop the function is in
    IF ( (Dims(1).LE.0).OR.(Dims(2).LE.0) ) THEN
        WRITE(*,*) "ERROR: matrix dimensions CAN NOT BE negative or 0"
        STOP
    END IF

    ALLOCATE( Init_matr%Elem(Dims(1),Dims(2)) )
    DO ii=0,Dims(1)
        DO jj=0,Dims(2)
            Init_matr%Elem(ii,jj)=Value_
        END DO
    END DO

    ! Initializing trace and determinant to 0
    Init_matr%Trace = (0d0, 0d0)
    Init_matr%Det = (0d0, 0d0)

    RETURN

END FUNCTION Init_matr
```

The subroutine to print the matrix was done as follows:

- Since the matrix to print can be large, the choice was to print dimensions, trace and determinant first; after those, matrix elements are printed;
- The elements are printed row-wise (one matrix row for each line of file); if the matrix is too large, this way of representing it can become sub-optimal: however, when the matrix is too big its representation in a human understanable way can be generally hard;
- The matrix is printed into a file which name is passed to the function in the code (it can be implementend in the program to ask the name of the file).

```
! Subroutine to print matrix
SUBROUTINE Dump_Matr(Matrix_, File_name)
    ! Matrix_  : TYPE DMATRIX

    IMPLICIT NONE

    TYPE(DMATRIX), INTENT(IN) :: Matrix_
    CHARACTER(LEN=*), INTENT(IN) :: File_name
    INTEGER*8 :: ii

    ! Opening file (ID=51)
    OPEN(UNIT=51,FILE=File_name,STATUS="unknown")

    WRITE(51,*) "Matrix dimensions are: "
    WRITE(51,*) Matrix_%Dims
    WRITE(51,*) "Matrix trace is: "
    WRITE(51,*) Matrix_%Trace
    WRITE(51,*) "Matrix determinant is: "
    WRITE(51,*) Matrix_%Det

    WRITE(51,*) "Matrix elements are: "

    DO ii=1,Matrix_%Dims(1)
        WRITE(51,*) Matrix_%Elem(ii,:)
    END DO

    CLOSE(51)

END SUBROUTINE Dump_Matr
```

In the end, a test program was written: here a *DMATRIX* is initialized asking the user a complex value.
.*Trc.* operator was applied in both ways, saving the result in another *DMATRIX* or in a *COMPLEX*8* variable.
.*Adj.* operator was used directly in the function to print the *DMATRIX* into a file.

```
! Starting program
PROGRAM Ex2_MATRICES
    USE MATRICES
    IMPLICIT NONE

    TYPE(DMATRIX) My_try, Trace_
    INTEGER, DIMENSION(2) :: Dims
    COMPLEX*8 :: comp_Val
    INTEGER*8, DIMENSION(2) :: dim1


    WRITE(*,*) "Insert the 2 dimensions"
    READ(*,*) Dims(1), Dims(2)

    WRITE(*,*) "Insert the value to initialize the matrix"
    READ(*,*) comp_Val

    My_try = init(Dims, comp_Val)

    ! Want to overwrite the DMATRIX onto itself
    !My_try = .Trc.My_try

    Trace_ = .Trc.My_try
    My_try%Trace = .Trc.My_try%Elem

    WRITE(*,*) My_try%Elem
    WRITE(*,*) Trace_%Elem
    WRITE(*,*) Trace_%Trace
    WRITE(*,*) "Try_trace"
    WRITE(*,*) try_trace

    CALL Dump_Matr(My_try, "Matrix.txt")

    CALL Dump_Matr(.Adj.My_try, "Adjoint_Matrix.txt")

    STOP
END PROGRAM Ex2_MATRICES
```

Results

The program runs well, the files and the output are as expected. Some tries were made using non-square matrices and negative dimensions: the program worked as expected.

Self-evaluation

During this exercise I learned how to define new types in Fortran and how to handle them. I also learned how to handle operators and functions (very "simple" functions with few arguments) and their interfaces (both for operators and functions, although the definition of interfaces for functions can be needless in majority of cases). In my personal opinion, the definition of an operator should be properly chosen meaning that operators are different with respect to functions and their usage is peculiar; before understanding this fact, I was not able to distinguish them from functions properly and I made some mistakes that now are clear to me. An example is the fact that I wanted operators to modify their arguments, which is actually meaningless (for example, when using "+" operator the numbers involved are not really changed, but an additional number, the result, is generated).
Another thing that I learned is to write complex numbers, which require the format "(*real*,*immaginary*)".
For what concern the program, it worked well at the end, although it was corrected and simplified in some points to avoid problems such as overwritting the whole *DMATRIX*, (that I was not able to solve: the idea was to update it, but I found an alternative) or avoiding some *memory segmentation* errors that arose during the previous executions; these errors probably were due to a not correct allocation or a bad handling of allocated elements of the *DMATRIX* (the test program now works, but I can not be sure there will be no more memory errors during future executions).
Finally, in my opinion the functions can still be improved, for example checking also the type of the dimensions (which should be integer) or asking for more user input.