Exercise 2: *Theory and implementation of derived data types in Fortran 90.*

Iriarte Delfina (1231682)

Quantum Information

October 20, 2020

Abstract

In this report, a **derived-data type** was successfully implemented in FORTRAN 90. In particular, a `DOUBLE COMPLEX MATRIX` type was built with the following components: elements, dimension, trace and determinant. The trace and the adjoint of the matrix were successfully computed by creating their correspondent functions as well as their **interface blocks**. Finally, all the results obtained of the matrix were saved into a TXT file by implementing a `SUBROUTINE`.

1 Introduction

Main new features were introduce by FORTRAN 90/95 such as the possibility to define special data types, known as **derived–data types**, and **interface block**. Users can create new types for their own purpose apart from the available intrinsic data type such as `INTEGER` or `REAL`. In fact, derived data types enable the building of more sophisticated types than intrinsic ones.

In this report, a `MODULE` is implemented in order to create a matrix type as well as functions and subroutines that operate on the matrix object. We also show how to access elements of a derived type using the `%` command. In particular, a `DOUBLE COMPLEX MATRIX` type is built with the following components: elements, dimension, trace and determinant. The initialization of the matrix is performed by a `SUBROUTINE`. Moreover, two functions are implemented to compute the **trace** and the **adjoint** of a matrix with their correspondent `INTERFACES`. Finally, another `SUBROUTINE` is built in order to write the information of the matrix into a TXT file.

Before going inside the details of the code, the formalism of the trace and the adjoint of a matrix are given. Given an operator A, we can define the **trace** as the sum of the diagonal elements, namely:

$$\text{Tr}\{A\} = \sum_{j=1}^{N} A_{jj} = \sum_{j} \langle v_j | A | v_j \rangle \tag{1}$$

and the **adjoint** $A^{\dagger}$ as an operator acting on the dual space with the property: $\langle \phi | \psi \rangle = \langle \psi | \phi \rangle^{*}$ where $*$ indicates the complex conjugate. In matrix representation this means that the adjoint of an operator is the conjugate transpose of that operator:

$$A^{\dagger}_{k,j} = \langle k | A^{\dagger} | k \rangle = \langle j | A | k \rangle^{*} = A^{*}_{j,k}$$

2 Code development

In this project, two modules were implemented. The first one is mainly for precision purposes and it is shown in Listing 1. The implementation among the project is simply by POINTING the precision that we would like, explicitly `use precision, pr=>dp`.

```fortran
module precision
    implicit none
    integer, parameter :: sp = kind(1.0)
    integer, parameter :: dp = kind(1.d0)
    !integer, parameter :: qp = kind(1.q0)
end module precision
```

Listing 1. Module precision

The main module of our project includes the implementation of the `DOUBLE COMPLEX MATRIX` type, two functions and two subroutines as it was already mentioned. In order to use an object of a derived-data type, we must first define the form of it. In Listing 2, the implementation of the derived-data types is shown. As it can be seen, the components of the `DOUBLE COMPLEX MATRIX` are given by its:

- **Elements**: `mat_elements(:,:)`;
- **Rows and columns**: `n_row, n_col`;
- **Trace**: `mat_trace`;
- **Determinant**: `mat_det`.

```fortran
    type matrix
        double complex, allocatable :: mat_elements(: , :)
        integer(pr) :: n_row, n_col
        double complex :: mat_trace
        double complex :: mat_det
    end type matrix
```

Listing 2. Matrix type

The initialization of the matrix is given by the `SUBROUTINE initialization (mat, nrow, ncol)` shown in Listing 3. The subroutine takes as input the matrix, the number of rows and columns and allocate them. This is done by implementing the function `allocate (mat%mat_elements(nrow, ncol))`. The matrix is built increasing the elements of the row when increasing the elements of the columns and viceversa. Once the matrix is initialized, it is printed in the bash. As a final remark, notice that in order to acces to the elements of the `type (matrix)` we used the % command.

```fortran
        SUBROUTINE initialization (mat, nrow, ncol)
        use precision, pr=>dp
        implicit none
        integer(pr) :: ii, jj
        integer(pr), intent(in) :: nrow, ncol
        type (matrix), intent(inout) :: mat
        mat%n_row = nrow
        mat%n_col = ncol

        allocate (mat%mat_elements(nrow, ncol))
        do ii = 1, nrow
            do jj = 1, ncol
            mat%mat_elements(ii, jj) = CMPLX(1._pr*ii,1_pr *jj)
            end do
        end do
```

```
       100 format (*('('sf6.2xspf6.2x'i)':x))
       do ii = 1, nrow
            write(*,100) (mat%mat_elements(ii, jj), jj = 1, ncol)
        end do
        mat%mat_trace = (0._pr, 0._pr)
        mat%mat_det = (0._pr, 0._pr)
    END SUBROUTINE
```

Listing 3. Initialization of the matrix

In Listing 4, a function is built in order to compute the trace of the matrix. Initially, the trace `diagonal_sum` is set to 0 and then, if the matrix is square, we sum each of the elements of the diagonal the matrix as given in equation 1.

```
    function diagonal_sum(matt)
        use precision, pr=>dp
        implicit none
        type(matrix), intent(in) :: matt
        integer(pr) :: kk
        double complex diagonal_sum
        diagonal_sum = (0._pr, 0._pr)

        !the trace is defined for square matrices only
        if (abs(matt%n_row - matt%n_col) < 0.5) then
            do kk = 1, matt%n_row
                diagonal_sum = diagonal_sum + matt%mat_elements(kk,kk)
            end do
        end if
    end function
```

Listing 4. Trace function

The computation of the adjoint of the matrix is shown in Listing 5. The function takes as input the matrix and returns their correspondent adjoint. In order to compute it, the transpose is computed: first, the number of rows and columns are inverted for the adjoint matrix as well as their elements are flipped. At the same time, the complex conjugate is computed implementing the function `DCONJG` provided by FORTAN (`D` holds for DOUBLE PRECISION). Finally, as it was done with the initialization, the adjacent matrix is printed in the bash.

```
        function adjoin(matrix_a)
        use precision, pr=>dp
        implicit none
        type(matrix), intent(in) :: matrix_a
        type(matrix) :: adjoin
        integer (pr) :: xx, yy
        adjoin%n_col = matrix_a%n_row
        adjoin%n_row = matrix_a%n_col
        allocate(adjoin%mat_elements(adjoin%n_row,  adjoin%n_col))

        do xx = 1, matrix_a%n_row
            do yy = 1, matrix_a%n_col
                adjoin%mat_elements(xx,yy) = dconjg(matrix_a%mat_elements(yy,xx))
            end do
        end do
    end function
```

Listing 5. Adjoint function

The INTERFACE statement of the functions `diagonal_sum` and `adjoin` are shown in Listing 6. As it can be seen, the references that invoke the two functions are specified by some *dummy* *arguments*, respectively `.Tr.` and `.Adj.`.

```
    interface operator (.Adj.)
    module procedure adjoin
     end interface

     interface operator (.Tr.)
     module procedure diagonal_sum
     end interface
```

Listing 6. Interface block

The `SUBROUTINE writing_file(txt_name, mat)` takes as input a matrix and a file name and save it into the correspondent TXT file. The implementation is shown in Listing 7.

```
SUBROUTINE writing_file(txt_name, mat)
        implicit none
        integer, parameter :: pr = kind(1.d0)
        character(:), allocatable :: txt_name
        type (matrix), intent(in) :: mat
        integer(pr) ::ii, jj
        open(11, file = txt_name, status = 'unknown', action = 'write')
        write(11, 120)
        120 format ('#' 25x, 'Matrix values ',/)
        100 format (*('('sf6.2xspf6.2x'i)':x))
        do ii = 1, mat%n_row
             write(11,100) (mat%mat_elements(ii, jj), jj = 1, mat%n_col)
        end do
        write(11, *) 'Number of rows:', mat%n_row
        write(11, *) 'Number of cols:', mat%n_col
        write(11, *) 'Trace:', mat%mat_trace
    end SUBROUTINE
```

Listing 7. Subroutine that writes the results obtained into a file.

Finally, the main program is shown in Listing 8. It creates two `DOUBLE COMPLEX MATRIX` types (one for the original matrix and the other for the adjacent one) and ask for the user to introduce the number of rows and columns that desire. The initialization of the matrix is performing by `CALLING` the correspondent SUBROUTINE as well as the loading of the results into the data file. Recall that the trace of the matrices are a element of the derived type, hence we need to access it by applying %. The result of the trace will then be given by invoking the correspondent interface, explicitly `matriz%mat_trace = .Tr.(matriz)`. The same holds for the calculation of the adjacent function, were the interface block is called, `matriz_1 = .Adj.(matriz)`

```
program main
    use matmodule
    IMPLICIT NONE
    type(matrix) :: matriz
    type(matrix) :: matriz_1
    integer(pr) :: row, col, i, j
    double complex :: trace
    character(:), allocatable :: file_1, file_2

    file_1= "matrix.txt"
    file_2= "matrix_adjacente.txt"

    write(*,*) "number of rows desire:"
    read(*,*) row
    write(*,*) "number of cols desire:"
    read(*,*) col
```

```fortran

    call initialization (matriz, row, col)

    matriz%mat_trace =  .Tr.(matriz)
    102 format  (*('('sf6.2xspf6.2x'i)':x))

    write(*,102)  matriz%mat_trace

    matriz_1 = .Adj.(matriz)
    matriz_1%mat_trace= .Tr.(matriz_1)
    write(*,102)  matriz_1%mat_trace

    call writing_file(file_1, matriz)
    call writing_file(file_2, matriz_1)
END PROGRAM
```

Listing 8. Main program

3 Results

To test the correctness of the program, the results of the TXT file are provided. It can be seen that the adjacent of the original matrix is the correct one as we wanted.

data.txt

```
#                          Original Matrix values
  1.00  +1.00 i  1.00  +2.00 i  1.00  +3.00 i  1.00  +4.00 i  1.00  +5.00 i  1.00  +6.00 i  1.00  +7.00 i
  2.00  +1.00 i  2.00  +2.00 i  2.00  +3.00 i  2.00  +4.00 i  2.00  +5.00 i  2.00  +6.00 i  2.00  +7.00 i
  3.00  +1.00 i  3.00  +2.00 i  3.00  +3.00 i  3.00  +4.00 i  3.00  +5.00 i  3.00  +6.00 i  3.00  +7.00 i
  4.00  +1.00 i  4.00  +2.00 i  4.00  +3.00 i  4.00  +4.00 i  4.00  +5.00 i  4.00  +6.00 i  4.00  +7.00 i
  5.00  +1.00 i  5.00  +2.00 i  5.00  +3.00 i  5.00  +4.00 i  5.00  +5.00 i  5.00  +6.00 i  5.00  +7.00 i
  6.00  +1.00 i  6.00  +2.00 i  6.00  +3.00 i  6.00  +4.00 i  6.00  +5.00 i  6.00  +6.00 i  6.00  +7.00 i
  7.00  +1.00 i  7.00  +2.00 i  7.00  +3.00 i  7.00  +4.00 i  7.00  +5.00 i  7.00  +6.00 i  7.00  +7.00 i
 Number of rows:                    7
 Number of cols:                    7
 Trace:               28.000000000000000,28.000000000000000
```

data.txt

```
#                          Adjacent Matrix values
  1.00  -1.00 i  2.00  -1.00 i  3.00  -1.00 i  4.00  -1.00 i  5.00  -1.00 i  6.00  -1.00 i  7.00  -1.00 i
  1.00  -2.00 i  2.00  -2.00 i  3.00  -2.00 i  4.00  -2.00 i  5.00  -2.00 i  6.00  -2.00 i  7.00  -2.00 i
  1.00  -3.00 i  2.00  -3.00 i  3.00  -3.00 i  4.00  -3.00 i  5.00  -3.00 i  6.00  -3.00 i  7.00  -3.00 i
  1.00  -4.00 i  2.00  -4.00 i  3.00  -4.00 i  4.00  -4.00 i  5.00  -4.00 i  6.00  -4.00 i  7.00  -4.00 i
  1.00  -5.00 i  2.00  -5.00 i  3.00  -5.00 i  4.00  -5.00 i  5.00  -5.00 i  6.00  -5.00 i  7.00  -5.00 i
  1.00  -6.00 i  2.00  -6.00 i  3.00  -6.00 i  4.00  -6.00 i  5.00  -6.00 i  6.00  -6.00 i  7.00  -6.00 i
  1.00  -7.00 i  2.00  -7.00 i  3.00  -7.00 i  4.00  -7.00 i  5.00  -7.00 i  6.00  -7.00 i  7.00  -7.00 i
 Number of rows:                    7
 Number of cols:                    7
 Trace:              28.000000000000000,-28.000000000000000
```

4 Conclusion

The implementation in FORTRAN 90 of a **derived-data type** was successfully performed. It can be seen that the FORTRAN program becomes more readable and modular with sensible use of derived data types. At the same time, **interface blocks** provides a smart way to implement functions.