UNIVERSITY OF PADUA
PHYSICS AND ASTRONOMY DEPARTMENT

Homework #2

Student name: *Beatrice Segalini* – 1234430

Course: *Quantum Information and Computing 2020* – Professor: *S. Montangero*
Due date: *October 20th, 2020*

Abstract

In this report, the description of a `Fortran90` program for handling matrices is displayed. The code contains a MODULE with a TYPE including the components: Matrix elements, Matrix Dimensions, Matrix Trace, and Matrix Determinant. It also includes functions to initialise with random complex values a matrix of the newly defined type, and to compute its TRACE and ADJOINT. A print and a write-on-file subroutines are also defined.

Theory

In this exercise, the focus is on matrix manipulation. In particular, two main operators are exploited and implemented in the code: TRACE and ADJOINT.

Let A be a square matrix of size $N \times N$ whose elements are denoted as $a_{i,j}$. The *adjoint* matrix $A^\dagger$ is defined as the transpose and complex conjugate of A, i.e. the matrix whose elements are:

$$a^\dagger_{i,j} = a^\star_{j,i} \tag{1}$$

Using the same notation, the *trace* is defined as the sum of the diagonal elements of A:

$$\mathrm{Tr}(A) = \sum_{i=1}^{N} a_{i,i} \tag{2}$$

In particular, the following property holds:

$$\mathrm{Tr}(A) = \mathrm{Tr}^* \left(A^\dagger\right) \tag{3}$$

Code development

At first, the module `MATRICES` is defined. It includes all the required features of the code, namely:

1. the `DMATRIX` type;
2. the `InitMatrix` function, to initialise a `DMATRIX` with some random complex values with $\Re$, $\Im \in (0, 1)$;
3. the `MatTrace` and the `MatAdjoint` functions, to compute the trace and the adjoint of a `DMATRIX`, and their respective operators, `.Trace.` and `.Adj.`;
4. the output subroutines, `printmatrix` and `writematrix`.

Then, everything is tested in the progam `testDMATRIX`.

The type DMATRIX. As requested, this user-defined type includes matrix shape in a $2-$dimensional vector `N`, the matrix elements $a_{i,j}$ in the vector `elements` and its `trace` and `determinant`.

InitMatrix. This function takes as input the number of rows and columns of a matrix and return a DMATRIX.

The entries of the matrix are generated by creating 2 matrices of random double-precision real numbers (`xx, yy`) and combining them with the `Fortran` intrinsic function `CMPLX`.

The trace is computed with the `MatTrace` function, while the determinant is set to 1 and its calculation is not implemented.

MatTrace. This function takes as input a DMATRIX and returns a double complex `t`. The trace is computed only if the input matrix is square, otherwise it returns the value 0. The proper interface allows the application of the function as an operator named `.Trace.`.

MatAdjoint. It takes as input a DMATRIX and return a DMATRIX which is the adjoint of the input. The adjoint is computed with `Fortran` intrinsic functions. The proper interface allows the application of the function as an operator named `.Adj.`. Notice that the trace is computed exploiting the property 3.

printmatrix. This subroutine allows to print the input DMATRIX nicely and formatting the output: the complex numbers are written in the form $a_{i,j} = x + yi$ and the number of displayed digits is cut to 4.

writematrix. The `writematrix` subroutine uses the same format as `printmatrix`, but saves the output to a file, whose `filename` is decided by the user and taken as input. The output file contains the matrix, its trace and its determinant.

testDMATRIX. The program asks the user to insert the number of rows and columns of the matrix, which is then initialised with the previously described function `InitMatrix`. The `printmatrix` subroutine is called and the newly generated matrix is printed. The computation of the adjoint matrix and of the trace of both matrices is performed and results are shown on screen.

Finally, the user needs to write two strings representing the names of the output files, on which the subroutine `writematrix` will write the results.

Results

The code compiles and works properly. It has been tested with randomly generated matrices (initialised with the `InitMatrix` function) of different sizes and shapes, although not very big.

Self evaluation

The code can be improved in different ways, e.g.:

- implementing a check on the size of the input matrix: up to now, the code compiles and gives no errors but returns an empty matrix if the numbers $(0, 0)$ are chosen as sizes. Moreover, the program crashes if characters are inserted by the user as inputs.
- Coding a way to signal if the matrix is not square: this could help to distinguish cases in which the trace is actually 0 from cases when the trace does not exist.
- Optimizing the output such that it is more “automatic" and less user-dependent.
- Testing with different optimization flags of the compiler and with bigger matrices.

Apart from these further improvement, this exercise was useful to learn about managing a module and all its components, to apply functions and subroutines and how to control output formatting.