Quantum information and computing: Exercises report, week 2
Derived Types: complex matrices

Alessandro Lambertini
(Dated: October 20, 2020)

Through the exercises of this week, we take a little bit more confidence with the Fortran specific structure and tools, like modules, routines, interfaces and the syntax of the language in general. We have to define a new derived type inside a module with some properties and implement some specific functions and soubrutine designed specifically for this new type. Finally, we define some specific interfaces for the routines inside the module and we build a test program to verify the consistency and correctness of our work.

I. THEORY

A. Functions and subroutines

Subroutines and functions are the main tolls to build a Fortran program, like in many other languages. The main difference between them is that a function return a value through its name while subroutines don't. Functions must have at least one argument while subroutines may have none. The type of the function name must be declared both where it is defined and in the segment from which it is called. Functions are called simply through their name and arguments while to call a subroutine it is needed a CALL statement.
The formal syntax to implement a function is:

```fortran
function name_func(arg1, arg2, ...)
  ![declarations, including those for the arguments]
  ![executable statements]
end function name_func
```

Instead, for a subroutine we have:

```fortran
subroutine name_sub(arg1, arg2, ....)
  ![declarations, including those for the arguments]
  ![executable statements]
end subroutine name_sub
```

B. Modules

Modules enable the user to split the program in multiple files, which is useful for many reasons. First, in this way it is easier to handle big program and projects. Second, it is possible to test and debug only 'small' pieces of code at a time. Third, it is possible to reuse functions and routines thought for other programs.
The formal syntax to implement a module is:

```fortran
module name_module
  implicit none
  !statement declarations
  contains
  !subroutine and function definitions
  end module name_module
```

C. Derived types

A derived data type is a data structure defined by the programmer, which consists of other data types (including other derived data type). Derived data types enable grouping of data to form logical objects, and a sensible use of them would made a Fortran program more readable and modular.

The formal syntax to define a new derived type, and to access its components is:

```
type new_type
 !declarations of the components of type_name
end type new_type

type(new_type) :: var_1

var_1%component_1 = ...
var_1%component_2 = ...
```

Where (%) is the component selector character in Fortran.

D. Interfaces

An interface is all of the information that is needed to call a procedure. Fortran has always had implicit interfaces for all his intrinsic functions. From Fortran 90 on it is possible for the programmer to define explicit interfaces for his routines and made the compiler aware of the requirement needed to properly call them. The easiest way to make a procedure's interface explicit is to include the procedure in a module.
It can sometimes be useful to be able to refer to different procedures by the same name. This can be done by what is called 'overloading' procedure, that is basically grouping the different routines under the same interface.
The formal syntax to define an interface is:

```
\begin{lstlisting}
module name_module
  implicit none
  !statement declarations

  interface name_interface !operator(...)
    module procedure procedure_1, procedure_2,...
  end interface

  contains
  !procedure_1 ...
  !procedure_2 ...
  !...
  end module name_module
```

II. CODE DEVELOPMENT

The first thing required by the exercise is to create a module which contains a double complex matrix derived type that includes the components: Matrix elements, Matrix dimension, Matrix trace, matrix determinant.
In order to do this i created an .f90 file named *matrix_module_try1.f90*, and i defined the new type in the following way:

```
module matrices
  implicit none
  private                                          !define what is accesible
  public dmatrix, AB, TRA, ADJ, write_onf_dmat     !outside the module


  !derived type declaration
  type dmatrix
    integer, dimension(2) :: dim
    complex(kind=8), dimension(:,:), allocatable :: elements
    real(kind=8) :: trace, determinant
  end type dmatrix
```

Where all the components are in double precision exept the %dim component which is an array with two single precision integer expected inside.
Secondly, we are requested to implement a function or a subroutine to initialize the new type just defined. I thought that a function with the dimensions of the matrix as arguments was a good choice, so I implemented it in the following:

```
!function to initialize the new type. The commented parts are a first version
!initialization with random number.
function Initialize_dmatrix(a,b)
  integer(kind=4) :: i,j
  integer(kind=4) :: a,b
  type(dmatrix) :: Initialize_dmatrix

  Initialize_dmatrix%dim(1) = a
  Initialize_dmatrix%dim(2) = b

  allocate(Initialize_dmatrix%elements(a,b))

  do i = 1, a
    do j = 1, b
      Initialize_dmatrix%elements(i,j) = (0,0)
    end do
  end do

  Initialize_dmatrix%trace = 0
  Initialize_dmatrix%determinant = 0  !This will be always 0 untill we define
                                      !a funtion to calculate the determinant
end function Initialize_dmatrix
```

As can be seen from the code, given the dimensions of our matrix, the function initialize all the elements, the trace and the determinant to 0. In the file are present some commented lines of code, not reported here, that referred to a previous version of the function thought to initialize the matrix with random numbers as elements. I changed my mind because it was not requested to explicitly compute the determinant and so this kind of initialization is more simply and more correct. Perhaps, those lines will be useful in future, so i decided to not delete them.
After that, we have to define two functions or subroutine to compute the trace and the transpose, conjugate matrix of a matrix defined as a 'dmatrix':

```
!function to compute the trace of a dmatrix.
function trace(MM)
  type(dmatrix) :: MM
  integer(kind=4) :: i
  real(kind=8) :: trace

  trace = 0

  if (abs(MM%dim(1) - MM%dim(2))>1e-1) then
    !write(*,*) 'this is not a square matrix'
    trace = 999999999999999
  else
    do i = 1, MM%dim(1)
      trace = trace + MM%elements(i,i)
    end do
  end if

end function trace
```

This is the function for the trace, with a conditional statement to check if the argument is a square matrix or not. The commented line of code referred to a version of the module in which was not yet present the subroutine that writes the results on a file.

In the following is reported the function to calculate the transpose adjugate of a 'dmatrix', where I essentialy use two intrinsic function of Fortran to complete the task.

```
!Function to compute the conjugate transpose matrix of a dmatrix
function Adjoint(MM)
  type(dmatrix) :: MM, Adjoint

  Adjoint%dim(1) = MM%dim(2)
  Adjoint%dim(2) = MM%dim(1)

  allocate(Adjoint%elements(Adjoint%dim(1),Adjoint%dim(2)))
  Adjoint%elements = conjg(transpose(MM%elements))

  adjoint%trace = MM%trace
  adjoint% determinant = 0  !still to be defined the function to compute
                            !the determinant
end function Adjoint
```

Moreover, we have to define the correspondent interfaces for the previously defined functions, which must appear in the declaration part of the module.
The last routine to insert in the module is a subroutine that writes the 'dmatrix' on a file in a readable form. I implemented it in the following way:

```
!subroutine useful for write on a file a dmatrix
subroutine write_onf_dmat (MM,filename)
  type(dmatrix) :: MM
  character(len=*) :: filename
  integer :: i, j

  open (unit = 1, file = filename)
  Write(1,*) '# rows:', MM%dim(1)
  Write(1,*) '# columns:', MM%dim(2)
  write(1, *) ''

  write(1,*) 'Matrix:'
  write(1,*) ''
  do i = 1, MM%dim(1)
    do j = 1, MM%dim(2)
      write(1, '(ES0.0,"+i","(",ES0.0,")",3X)', advance='no') MM%elements(i,j)
    end do
    write(1, *) ''
  end do

  write(1,*) ''
  if (abs(MM%dim(1) - MM%dim(2))>1e-1) then
    write(1,*) 'Trace: undefined'
    write(1,*) 'Determinant: undefined'
  else
    write(1,*) 'Trace:', MM%trace
    write(1,*) 'Determinant:', MM%determinant
  end if
  close(1)
end subroutine write_onf_dmat
```

This subroutines takes as arguments the matrix to be write on file and the filename which is created from scratch. It has been a little bit challenging figuring out the right options for the 'write' statement in order to give a nice format to the elements of the matrix.
Finally, we have to include the module in a test program and verify its behavior. I implemented a simple program, in a file called *matrix_mod_test.f90*, that initialize the dmatrix with some random numbers between 0 and 1 both for the real and the immaginary part. Then, it calculates the trace and the transpose conjugate matrix of the one just initialized, finally it prints this two matrices and their properties into two different files.

The code of the test program is the following:

```
program matrix_mod_test
  use matrices
  implicit none

  type(dmatrix) :: mat_1, mat_1_t
  integer :: i,j
  complex(kind=8) :: ij
  real(kind=8) :: Re_z, Im_z, tr

  mat_1 = AB(4,4)    !initialize a 4X4 dmatrix

  do i = 1, mat_1%dim(1)
    do j = 1, mat_1%dim(2)
      CALL RANDOM_NUMBER(Re_z)
      CALL RANDOM_NUMBER(Im_z)       !fill the matrix with random numbers
      ij = cmplx(Re_z,Im_z)
      mat_1%elements(i,j) = ij
    end do
  end do



  mat_1%trace = TRA(mat_1) !calculate the trace
  mat_1%determinant = 0    !still not defined a function to calculate the determinant

  mat_1_t = ADJ(mat_1)    !calculate the transpose conjugate matrix

  call write_onf_dmat(mat_1,'mat_1.txt')    !write mat_1 into 'mat_1.txt'
  call write_onf_dmat(mat_1_t,'mat_2.txt')

  deallocate(mat_1%elements)    !deallocate memory
  deallocate(mat_1_t%elements)

end program matrix_mod_test
```

III. RESULTS

The code written seems to work properly. The results produced with my machine are two files named *mat_1.txt* and *mat_1_t.txt* where one can find, hopefully in a readable form, the dimensions, the elements, the trace and the (not yet computed) determinant of the matrices.
All the files produced working on the code are located in the workink folder *Ex2-Lambertini-CODE*. In the two main files, the module and the test program, are present comments about the compilation options used to compile the two file with the gfortran compiler. This report will be placed in the folder *Ex2-Lambertini-REPORT*, and both this two folders are located in *Ex2-Lambertini* that will be uploaded to Moodle. I think it's a good way of organizing the delivery of this and the future works, if not please let me know.

IV. SELF-EVALUATION

I think the main objectives of the exercise are achieved. However, I think the structure of the report is not perfect, perhaps it will become clearer to me what is required by working on the next assignments. Also, it is the first approach to Fortran for me and I'm sure the code is not elegant and I have probably missed something.