MASTER'S DEGREE IN PHYSICS
Academic Year 2020-2021

QUANTUM INFORMATION

Student: Giorgio Palermo
Student ID: 1238258
Date: October 20, 2020

EXERCISE 2

In this report I will review my solution to EX2, which is about the definition of new types, functions, subroutines and interfaces.

Theory

I based my solution of the proposed exercise on the definition of the `type`, `function`, `subroutine` and `interface` constructs reviewed in class.

Code Development

The basic brick of this program is the `dmatrix` type, which I defined as a new type containing a `double complex` matrix and some of its properties: shape, track and determinant.

```
type dmatrix
    integer, dimension(2) :: N = (/ 0,0 /)
    double complex, dimension(:,:), allocatable :: elem
    double complex :: Trace
    double complex :: Det
end type dmatrix
```

The `InitUni` function is a `type(dmatrix)` function that calls the `clarnv` LAPACK subroutine to fill the matrix (`dmatrix%elem`) with random complex numbers. Since `clarnv` only works on scalar or vectors, I implemented a cycle to fill the matrix; I chose to loop over columns because this is the fastest algorithm since the matrix is stored column-wise.
I decided that in my program the shape of a `dmatrix` has to be defined separately before the call to the initialization function, therefore I put a check at the beginning of it to verify that both dimensions are defined and positive.

```
function InitUni(dmat)
    ! Initializes a (m,n) complex matrix with
    ! real and imaginary part taken from [0,1]
    ! uniform distributions
    implicit none
    type(dmatrix), intent(in) :: dmat
    type(dmatrix) :: InitUni
    integer :: jj,sd=4
    integer, dimension(:), allocatable :: seed
    double complex, dimension(dmat%N(1),1) :: X

    if(dmat%N(1)<1 .or. dmat%N(2)<1) then
        ! Check for positive matrix shape
        print*, "*** ERROR in InitUni: matrix shape not defined"
        print*, "Program terminated"
        stop
    else
        call random_seed(size = sd)
        allocate(seed(sd))
        call random_seed(get=seed)
        do jj=1,dmat%N(2)
            ! '1' stands for uniform
            call clarnv(1, seed, 2*dmat%N(1), dmat%elem(:,jj) )
        end do
        InitUni = dmat
        return
    end if
end function InitUni
```

The `Tr` subroutine computes the trace summing over diagonal elements of a `dmatrix%elem` matrix given as input.

```
subroutine Tr(dmat)
    ! Computes the trace and assigns it
    ! to the "Trace" field of the input object
    ! of type dmatrix
    type(dmatrix) :: dmat
    integer ::ii
    if(dmat%N(1)==dmat%N(2) .and. dmat%N(1)>0 .and. dmat%N(2) >0) then
        do ii=1,dmat%N(1)
            dmat%Trace=dmat%Trace +dmat%elem(ii,ii)
        end do
        return
    else
        print*, "*** ERROR in Tr: matrix dimensions must be positive and equal"
        return
    end if
end subroutine Tr
```

`Adj` is a `type(dmatrix)` function which aim is to compute the transposed conjugate of a `type(dmatrix)` input. To do this it copies an input `dmatrix` type element into a local new variable and computes the adjoint using the intrinsic elemental function `conjg()`; the transposition is then performed using the intrinsic `transpose()` function.

```
function Adj(dmat)
    ! computes the adjoint and passes it
    ! as output
    type(dmatrix),intent(in) :: dmat
    type(dmatrix) ::Adj
    Adj%Trace=conjg(dmat%Trace)
    Adj%Det=conjg(dmat%Det)
    Adj%N=(/ dmat%N(2), dmat%N(1) /)
    allocate(Adj%elem(Adj%N(1),Adj%N(2)))
    Adj%elem=dmat%elem
    Adj%elem=conjg(Adj%elem)
    Adj%elem=transpose(Adj%elem)
    return
end function Adj
```

I assigned the `Adj` and the `InitUni` functions to two interface operators: `.Adj.` and `.Init.`.

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

interface operator (.Init.)
    module procedure InitUni
end interface
```

All these functions, subroutines and interfaces are defined inside a module.

Results

All the functions and subroutines are tested in a simple program, `DMatrixCODE`, which calls all the above mentioned functions and operators. More specifically, it defines and initializes a new `dmatrix` type variable, computes its trace and adjoint and writes both the matrix and its adjoint to file, using the subroutine `MatToFile` which I implemented to do this job.

```
program DMatrixCODE
    use stuff
    implicit none
    type(dmatrix) :: dmat, dmat1
    integer, dimension(2) ::shape = (/ 2, 2 /)

    write(*,'(A,/,A,/)') "","     *** DMatrixCODE.f03 - Complex matrix manipulation ***
     "

    dmat%N=shape
    allocate(dmat%elem(dmat%N(1),dmat%N(2)))
    allocate(dmat1%elem(dmat%N(1),dmat%N(2)))
    print*, "Initializing matrix..."
```

```
    dmat = .init.dmat
    print*, "Computing Trace, Adjoint..."
    call Tr(dmat)
    dmat1 = .Adj.dmat
    print*, "Writing results to file..."
    call MatToFile(dmat,"matrix")
    call MatToFile(dmat1,"matrix_conj")

    deallocate(dmat%elem,dmat1%elem)
    write(*,'(/,A)') "     *** End of the program"
end program
```

Hereafter are reported two typical outputs for a randomly initiated 2×2 matrix and its adjoint.

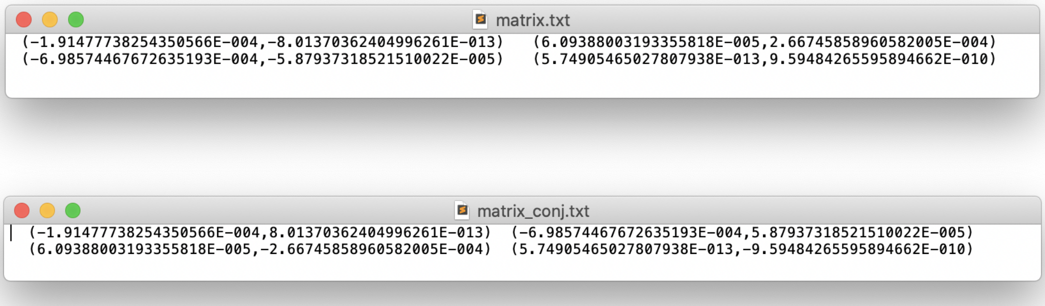

Figure 1: Output files examples

Self evaluation

Writing this exercise I learned how to define new types, functions, subroutines and interface operators; I also learned to call external LAPACK functions and to compile the code including the linear algebra library.
I wonder if in `Tr()` function is sufficient to check for the dimensions of the matrix to be positive or it would be recommendable to check if the memory for the `dmatrix%elem` is already allocated, in order to avoid errors.