MASTER'S DEGREE IN PHYSICS
Academic Year 2020-2021

QUANTUM INFORMATION

Student: Giorgio Palermo
Student ID: 1238258
Date: October 26, 2020

EXERCISE 3

In this report I will describe how I wrote a debug module which helps to perform some consistency checks on variables to detect errors; later I will describe how I improved the readability of my code using comments and documentation.

Code Development

The module I wrote is named `debug` and its purpose is to provide some functions to be inserted into a program and used for debugging.

The run of a scientific program does not provide for live user interaction: this is due to the fact that many tasks require to repeat the same operation a lot of times and a live output of the results on screen or the request of manual confirmation at each iteration would enlarge the total computation time. On the contrary, a debug software must interact with the user to display the results of its job: errors, warnings, anomalous values. Considering both these facts, I chose to build debug functions with an input variable to be used as a switch for the debug process. I chose this variable, which in the code below is named `activation`, to be an integer: this is because I want to be able, in the future, to expand my debug functions and select, for example, different results to be printed for different values of `activation`.
For each function I included two optional arguments; the first, `message`, is an optional message to be printed to indicate the purpose of the check (e.g.: "Checking energy conservation..."); the second, `print`, toggles the message containing the kind of check that is performed and the result.
I chose these to be `function` and not `subroutine`, because I want to be able, in the future, to expand this debug software to store all the results of the checks (aka the results of the functions) on file; this could help to debug problems with variables changing their values at each iteration of a loop.

I implemented a total of five functions: the first one is displayed below and its aim is to check if two matrices have the correct shape to be row-by-column multiplied; the function, as an option, can check if a third matrix has the correct dimensions to store the result.

```
function CheckDim(activation,A,B,C,message, print)
    ! Checks dimensions for matrix multiplication
    ! 0 = OK, 1 = ERROR
    integer :: activation, a2, b1, c1, c2
    double precision, dimension(:,:), allocatable :: A,B
    double precision, dimension(:,:), allocatable, optional :: C
    character(*), optional :: message
    integer, optional ::print
    logical :: CheckDim
    if(activation==1) then
        a2=size(A,2)
        b1=size(B,1)
        if(present(C))then
            c1=size(C,1)
            c2=size(C,2)
            CheckDim = .not.((a2==b1).and.(c1==a2).and.(c2==b1))
        else
            CheckDim = .not.((a2==b1))
        end if
        if(present(message)) then
            message = "MESSAGE *** "  // message
            write(*,*) message
        end if
        if(present(print)) then
            write(*,*) "CHK:  Matrix multiplication shape check (double): ", CheckDim
        end if
```

```
    else
        return
    end if
end function CheckDim
```

The previous function works for real, double precision numbers: since it is not uncommon to work also with integers, I implemented another function to perform the same check, but for integer numbers:

```
function CheckDimInt(activation,A,B,C, message, print)
    ! Checks dimensions for matrix multiplication
    ! 0 = OK, 1 = ERROR
    integer :: activation, a2, b1, c1, c2
    integer, dimension(:,:), allocatable :: A,B
    integer, dimension(:,:), allocatable, optional :: C
    logical :: CheckDimInt
    character(*), optional :: message
    integer, optional :: print
    if(activation==1)then
        a2=size(A,2)
        b1=size(B,1)
        if(present(C))then
            c1=size(C,1)
            c2=size(C,2)
            CheckDimInt = .not.((a2==b1).and.(c1==a2).and.(c2==b1))
        else
            CheckDimInt = .not.((a2==b1))
        end if
        if(present(message)) then
            message = "DEBUG *** " // message
            write(*,*) message
        end if
        if(present(print)) then
            write(*,*) "DEBUG *** Matrix multiplication shape check (integer): ",
    CheckDimInt
        end if
    else
        return
    end if
end function CheckDimInt
```

Another interesting feature for physical programs is to be able to check if two values (or arrays) are equal, for example for checking convergence, but also to ensure that conserved quantity do not change during the execution of a code due to numerical dissipation. Here is displayed a function that checks the equality of two arbitrarily sized arrays of integers:

```
function CheckEqInteger(activation,A,B,message,print)
    integer :: activation, a1, a2, b1, b2, ii,jj, err=0
    integer,optional :: print
    integer, dimension(:,:), allocatable :: A,B
    logical :: CheckEqInteger
    character(*), optional :: message
    CheckEqInteger = .false.
    if(activation/=1) then
        return
    else
        a1=size(A,1)
        a2=size(A,2)
        b1=size(B,1)
        b2=size(B,2)
        do jj=1,a2
            do ii=1,a1
                err=err + abs(A(ii,jj)-B(ii,jj))
            end do
        end do
        if(err/=0) then
            CheckEqInteger = .true.
        end if
        if(present(message))then
            message = "DEBUG *** " // message
            write(*,*) message
        end if
        write(*,*) "DEBUG *** Array inequality check (integer):", CheckEqInteger
        if(present(print))then
        write(*,*) "DEBUG *** Total error detected: ", err
```

```
        end if
    end if
end function CheckEqInteger
```

The check is performed computing a cumulative error, which is the difference of the two arrays computed element-wise; if this difference is different from zero, then an the function returns `.true.`.
This kind of check is not possible for real numbers, since their difference is almost always different from zero due to errors. Then a different method must be applied: in the following code I compute the cumulative error (this time normalized to the element of the second matrix which should be the reference value) and then I compare it to a threshold: a cumulative error greater than the threshold will set the output value to `.true.`. I chose this threshold to bel 10^{-5} : this is just a starting point to understand if two arrays are significantly different and then perform some more specific tests, that could be for example the computation of the error element by element.

```
function CheckEqDouble(activation,A,B,message, print)
    integer :: activation, a1, a2, b1, b2, ii,jj
    integer, optional :: print
    double precision err
    double precision, dimension(:,:), allocatable :: A,B
    logical :: CheckEqDouble
    character(*),optional :: message
    err = 0.0
    CheckEqDouble = .false.
    if(activation/=1) then
        return
    else
        a1=size(A,1)
        a2=size(A,2)
        b1=size(B,1)
        b2=size(B,2)
        do jj=1,a2
            do ii=1,a1
                err=err + abs((A(ii,jj)-B(ii,jj))/B(ii,jj))
            end do
        end do
        if(err>=1e-5*size(A,1)*size(A,2)) then
            CheckEqDouble = .true.
        end if
        if(present(message))then
            message = "MESSAGE *** "  // message
            write(*,*) message
        end if
        write(*,*) "CHK:  Array inequality check (double):", CheckEqDouble
        if(present(print))then
        write(*,*) "CHK:  Total error detected: ", err
        end if
    end if
end function CheckEqDouble
```

Another useful tool that I implemented for future analysis is the `CheckTrace` function, which checks if a matrix is square and then, if it is so, if the matrix has a positive trace.

Functions regarding matrix dimensions and equality are verified using a small program, called `DebugDebug.f03`:

```
program DebugDebug
    use Debug
implicit none
integer ii,jj, act
integer, dimension(:,:), allocatable :: A, B, C
double precision, dimension(:,:), allocatable :: A1, B1, C1
character(len=70) :: message
logical :: deb
act=1

... variables initialization ...

message = "Test on A,B integer, equal:"
deb= CheckDimInt(activation=act,A=A,B=B,message=message, print=1)
deb= CheckEqInteger(activation=act,A=A,B=B,print=1)
message = "Test on A,B integer, different"
B(2,2) = 65
deb= CheckDimInt(activation=act,A=A,B=B,message=message, print=1)
```

```
message = "Test on A,C integer, different dim"
deb= CheckDimInt(activation=act,A=A,B=C,message=message, print=1)
write(*,*)

message = "Test on A,B double, equal"
deb=CheckDim(activation=1,A=A1,B=B1, message=message,print=1)
deb=CheckEqDouble(act,A1,B1,print=1)
message = "Test on A,B double, different"
B(1,1)=.666
deb=CheckDim(activation=1,A=A1,B=B1, message=message,print=1)
deb=CheckEqDouble(act,A1,B1,print=1)
message = "Test on A,C double, different dim"
deb=CheckDim(activation=1,A=A1,B=C1, message=message,print=1)

end program DebugDebug
```

For the second part of the exercise, I reviewed the code of the program written for the first exercise `MatTest.f03`, writing a new version that I called `MatTest1.f03`. I added comments at each section of the code, that help understanding the content of the source file.
I added a long documentation: the first part consists of a header section, with all the essential info regarding the file: project, program name, author, date created, purpose and has a place to briefly recap changes done in the revision phase:

```
! ***********************************************************************
! Project           : Quantum information, Ex3
!
! Program name      : MatTest1.f03
!
! Author            : Giorgio Palermo
!
! Date created      : 20201007
!
! Purpose           : To test some operations with matrices
!
! Revision History  :
!
! Date        Author      Ref    Revision (Date in YYYYMMDD format)
!
! 20201020    G. Palermo         Debug subroutine implementation
! 20201021    G. Palermo         Change all reals to doubles
! 20201026    G. Palermo         New functions in Debug module,
!                                comments added
! ***********************************************************************
```

The second part is a brief recap of what the program does, with essential informations on how to run the program using the bash interface that is implemented i it; this part also contains information about the output format of the program. Here is reported the `MatTest1.f03` program description:

```
! *********   program MATTEST1    **************
! This program tests the computational performances of different matrix
! multiplication algorithms, by measuring CPU_TIME().
! The test is performed by multiplying two randomly generated square matrices
! using 1:LOOPMULT, 2:LOOPMULTCOLUMNS and 3:INTRINSICMULT.
! The program allows to either choose the matrix size in advance or perform an
! automatic test. Computation times are measured increasing the matrix size by
! 100 each step up to maximum size.

! I/O interface:
! The program is called via bash by
!     $ ./MatTest1.out [filename] [size]
! where both [filename] and [size] are optional arguments. The filename must be
! given without extension, since it is added automatically. [size] is the maximum
```

```
! matrix size that will be tested.
! If no arguments are provided:
!     - filename will be asked
!     - test will run up to default max_size (500)
!
! The output is given on file on four columns:
!
!     SIZE    T1[s]   T2[s]   T3[s]
```

Some information is provided also about the functions contained in `module functions` that are used in `MatTest1` to perform testing on matrices. In particular, for each subroutine, characteristics of the I/O arguments are reported, together with information regarding the aim of it, the methods implemented and the behavior of the subroutine in particular situations:

```
!     subroutine INTRINSICMULT(A,B,C)
!         Arguments:
!             A(:,:),B(:,:)    double precision,intent(in)
!             C(:,:)           double precision,intent(out),allocatable
!
!         Performs multiplication of double precision matrices
!         A,B and writes the result on C.
!         Multiplication is performed via intrinsic function MATMUL(A,B).
!
!         Preallocation is needed only for A,B (input).
!
!         The subroutine checks the match of size(A,2) and size(B,1)
!         and stops the execution of the program if the check fails.
```

In the program itself, together with the comments, checks have been implemented for the exit status of the opening (pre condition) and closing (post condition) procedures for the output file; a subroutine from the `debug` module is called at the beginning of each cycle to check if the matrix dimensions are incorrect and possibly stop the program.

Results and self evaluation

Through this homework I wrote and tested a debug module, that is able to perform different consistency checks in various situations and provide specific error messages for different errors. I also modified the look and some elements of my old `MatTest.f03` code to be more readable and to be understandable through the long documentation at the beginning.
This exercise has been useful to understand which elements of the code are crucial to be described to in the documentation for future use.