Quantum information and computing: Exercises report, week 3. Checkpoints and documentation

Alessandro Lambertini
(Dated: October 27, 2020)

Through the exercises of this week, we try to structure a piece of code with all the general good practices that must be followed when wants to write useful, reusable and comprehensible code. We have to implement a subroutine to be used as checkpoint for debugging, in which there must be a control on a logical variable and there could be an additional (optional) string of text, like an error message to be printed, and an (optional) variable, that could be the line under control. Moreover, we test the checkpoint subroutine in the program written for the exercise 3 of week 1, Test performance. Finally, we have to include, in the new version of the program, documentation, comments, pre and post conditions, error handling and, indeed, the checkpoints.

I. THEORY

A major part of programming is debugging and testing. The compiler itself helps to track down certain types of errors, but it is wrong to assume that a program is correct just because it compiles. Logging is a widely used practice to debug and test large programs. In it's simple form logging involves printing out information that can be useful to highlight problems, this process could span from completely manual to fully automatic. In this context we find the concept of checkpoints, routines that helps in debugging printing messages during the execution of the program. They are usually structured as routines that take as input a logical variable that decides if trigger the print or not, or which message has to be printed. In this way if there is a problem during the execution the programmer has the possibility to orients himself better in order to find the bug. These routines clearly can have additional arguments so that the messages can be more precise and useful. Another major tool in programs debugging is to set pre and post conditions. Pre and post conditions represent the state of our program in different moments, for example before entering and after exiting a specific routine. If these conditions are met, then the program keep being executed, if not then an error occurs and a message will be printed.

II. CODE DEVELOPMENT

A. checkpoints module

The first thing required by the exercises is to implement a subroutine to be used as a checkpoint for debugging. I started implementing a module named *debugging* in the file *checkpoints_module.f90* in which I wrote three subroutine that implement different checks on the code.
The first routine that is located in the module is the one that perform the checkpoint control:

```
!Subroutine useful for debugging that take a logical variable from another
!subroutines and decide whether to continue the execution or print some info
!and stop the program.

subroutine checkpoints (debug, sub_func_name, line)
  logical :: debug
  character(len=*) :: sub_func_name
  integer*4 :: line
  if (debug .eqv. .true.) then
    write(*,"(A,X,L,X,A,X,A,A,X,A,I0.0)") 'debug =', debug, &
    'for the func/subrt:', sub_func_name,'.','Line:',line
  else
    write(*,"(A,X,L,X,A,X,A,A,X,A,I0.0)") 'debug =', debug, &
    'for the func/subrt:', sub_func_name,'.','Line:',line
    stop
  end if

end subroutine checkpoints
```

This subroutine is thought to test other functions or subroutines. The argument *debug* is a logical variable whose value is determined as the result of operations performed in other routines called in the main program. The *sub_func_name* argument has to be a string representing the name of the routine tested and the *line* argument is an integer representing the line on which the control is performed. The subroutine print on screen a message and stop or not the program according to the value of the *debug* variable passed to it. In particular, it's printed on screen the value of *debug*, the name of the routine tested and the line of code where it's located. If the value of *debug* is *.true.* then the program continue to be executed, if instead the value is *.false.* the program will stop.
An example of the two output could be:

1. when *debug* is *.true.*

```
    debug = T for the func/subrt: dim_check. Line:34
```

2. when *debug* is *.false.*

```
    debug = F for the func/subrt: mat_mul_loop1. Line:50
```

To perform checkpoints with this subroutine it is clear that the routines used in the main program have to be written properly, including in them the optional logical argument *debug*. The other two subroutines present in the module are written in this spirit. They perform themselves a check, on the dimensionalitity of two matrices, to decide whether it is possible or not compute their row-column multiplication. I have written two subroutine to have the possibility to pass in input directly the two matrices or just their dimensions. Then, i have grouped them in a unique interface so that the program decide which one has to be executed according to the arguments.
Here is reported the subroutine that takes as arguments the two matrices, and the optional *debug* argument. The other subroutine performs exactly the same thing, but take as arguments two one-dimensional arrays which contain the dimensions of the two matrices.

```
!Subroutine to check if two matrices can be multiplied.
!It takes as arguments the two matrices and an optional logical variable
!useful to interact with checkpoints subroutine.
subroutine dimension_mul_check(mm_1,mm_2, debug)
  logical, optional :: debug
  real, dimension(:,:) :: mm_1, mm_2
  integer,dimension(2) :: col_1, row_2
  col_1 = shape(mm_1)
  row_2 = shape(mm_2)
  if (col_1(2) .ne. row_2(1)) then
    write(*,*) "This two matrices can't be multiplied, the program will stop"
    if (present(debug)) then
      debug = .false.
    else
      stop
    end if
  else
    write(*,*) "This two matrices can be multiplied"
    if (present(debug)) then
      debug = .true.
    end if
  end if
end subroutine dimension_mul_check
```

Finally i grouped them in the same interface:

```
module debugging
  implicit none
  private
  public dim_check, checkpoints

  interface dim_check
    module procedure dimension_mul_check
    module procedure dimensionality_check
  end interface
```

B. Test on exercise 3 of week 1

In order to test the module *debugging* on the third exercise of week 1 I had to adapt the code to the *checkpoints* subroutine. In particular, I re-wrote the program implementing each functionality as a subroutine, which is also a more correct approach in general. In each of the subroutine is present the optional logical argument *debug* that allows the interaction between them and the *checkpoints* subroutine through a logical global variable.

```
subroutine rand_init_mat(mm,dim_1,dim_2,debug)
  logical, optional :: debug
  integer :: dim_1, dim_2, i, j
  real(kind=8), dimension(:,:), allocatable :: mm

  if (present(debug)) then
    if (dim_1 .ge. 1 .and. dim_2 .ge. 1) then
      debug = .TRUE.
    else
      debug = .FALSE.
    end if
  end if


  allocate(mm(dim_1,dim_2))

  do i=1,dim_2
    do j=1,dim_1
      call RANDOM_NUMBER(mm(j,i))
      mm(j,i) = mm(j,i)*100
    end do
  end do
end subroutine rand_init_mat
```

This subroutine perform the initialization of a matrix with real random numbers in double precision. It also perform a control on the dimensions of the matrix, in particular it turn the value of the *debug* variable into *.false.* if the dimensions given in input are negative.
After that, I implemented the two subroutines that perform the multiplication with two different loops. They differ only for the order of two indices so here only one is reported, actually the slower one:

```
subroutine mat_mul_loop1(mm_1,mm_2, m_fin, debug)
  logical, optional :: debug
  integer :: i,j,k
  real(kind=8), dimension(:,:) :: mm_1, mm_2, m_fin
  integer, dimension(2) :: dimm_1, dimm_2, dimm_fin

  dimm_1 = shape(mm_1)
  dimm_2 = shape(mm_2)
  dimm_fin = shape(m_fin)

  if (present(debug)) then
    if (dimm_fin(1) .eq. dimm_1(1) .and. dimm_fin(2) .eq. dimm_2(2)) then
      debug = .TRUE.
    else
      debug = .FALSE.
    end if
  end if

  do i=1,dimm_1(1)
    do j=1,dimm_2(2)
      do k=1,dimm_1(2)
        m_fin(i,j)=m_fin(i,j)+mm_1(i,k)*mm_2(k,j)
      end do
```

```
    end do
  end do
end subroutine mat_mul_loop1
```

Here, we have as arguments the two matrices to be multiplied, the result matrix and the debug variable, which implements a control on the dimension of the result matrix.
It is clear that not all the control implemented in these routines are really useful, that is because I tried to emphasize the debugging and test phase as much as possible.
Here are reported some example of the call of a subroutine in the program and the call of the checkpoints subroutine:

```
!Initialize the two matrices to be multiplied and check if the dimensions of
!the two matrices are positive
call rand_init_mat(m_1,dim_m1(1),dim_m1(2),debug)
call checkpoints(debug, 'rand_init_mat', 23)
```

```
!check if the dimensions of the two matrices allows the multiplication
!if so allocate the dimensions for the final matrices and print a friendly message
!if not stop the program and print an error message

call dim_check(dim_m1,dim_m2, debug)
call checkpoints(debug, 'dim_check', 34)
```

```
call mat_mul_loop1(m_1,m_2,m_fin,debug)
!Matrix multiplication with the first loop
call checkpoints(debug, 'mat_mul_loop1',44)
!check if the dimensions of the resulting matrix are the expected ones
```

Finally, I report an example of the output given when the program is executed:

III. RESULTS

Both the *checkpoints_module.f90* and the file *ex_03.f90*, where it is implemented the main program, succesfully compile. Here is reported an example of the output produced by the code:

```
(base) MacBookProdiLambe:Ex-3-Lambertini-CODE lambe$ ./ex_03
debug = T for the func/subrt: rand_init_mat. Line:23
debug = T for the func/subrt: rand_init_mat. Line:26
This two matrices can be multiplied!
debug = T for the func/subrt: dim_check. Line:34
debug = T for the func/subrt: mat_mul_loop1. Line:44
Time loop 1   3.0000000000000512E-005 seconds
debug = T for the func/subrt: mat_mul_loop2. Line:54
Time loop 2   3.9999999999996635E-006 seconds
Intrinsic function   2.8000000000000247E-005 seconds
m_1:
57.799777543257903   99.101560106736528   75.137866419063386   79.337584404844421
98.533998274981599   85.159449173072659   99.771609182291130   85.511293866272780
73.473819234795641   27.421625603227106   57.608673091665693   4.865085325504658
27.878145501509223   33.193099725757676   3.191817623575643   22.469278767435519
m_2:
71.537878067080257   37.947549258559285   89.996123612925373
79.101558478059047   94.209452114188664   30.290142035159885
85.853831059922442   71.212553711029940   26.592174400271208
50.935229044610317   68.502259503123625   87.476477545074090
m_fin:
22465.913013495992345   22315.226730124901223   17141.807924672419176
26706.470521073249984   24724.656722729010653   21580.540509233796911
12618.983965332936350   9807.257763021289065   9400.484261578290898
6038.476924163429430   5951.504887720646366   5564.759464386936997
```

```
m_fin_2:
22465.913013495992345   22315.226730124901223   17141.807924672419176
26706.470521073249984   24724.656722729010653   21580.540509233796911
12618.983965332936350   9807.257763021289065   9400.484261578290898
6038.476924163429430   5951.504887720646366   5564.759464386936997
m_int:
22465.913013495992345   22315.226730124901223   17141.807924672419176
26706.470521073249984   24724.656722729007015   21580.540509233796911
12618.983965332936350   9807.257763021289065   9400.484261578290898
6038.476924163429430   5951.504887720647275   5564.759464386936088
```

All the files produced working on the code are located in the workink folder *Ex3-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.

IV. SELF-EVALUATION

I think the main objectives of the exercise are achieved. From the next assignment i will try to write the report while developing the code, because it seems to me that some conceptual passages could be explained better and more precisely in this way.