Quantum information and computation: homework 3
of Tommaso Tabarelli

Abstract

In this homework we are asked to create a subroutine to be used as checkpoint for debugging. It should have a control logical variable, some other optional variable and some string to be printed in case of errors to help handling the debug process.
We are also asked to implement such a process in the 3rd exercise of first week, adding comments, documentation, conditions and checkpoints.

Theory

During debugging, one looks for isseus that prevent the correct execution of a program. To do so, usually some strings are printed during the execution or some file with temporary results is created to highlight what piece of the program is not working as expected.

Code development

My choice was to implement 2 kinds of subroutine: one to be called in case of errors or warnings, which prints the error and usually stops program execution, while the other open some files in which store temporary results to be eventually re-collected by the program during the execution. Both subroutines were then adapted to be used in the 3rd exercise of first week.

The first subroutine has a very simple structure: it collects a *Debug* flag and a *Error* string. The flag states if activate or not the debug subroutine to print errors, while the string is a sort of identifier of the error: basing on which string is passed, the subroutine writes explicitely the error and then eventually stops program execution.

```
SUBROUTINE Print_error(Debug, Error)
   ! Subroutine used to DEBUG

   IMPLICIT NONE

   LOGICAL :: Debug
   CHARACTER(LEN=*) :: Error

   IF (Debug) THEN

         SELECT CASE (Error)

         CASE("Dim<=0")
            WRITE(*,*) "ERROR: dimensions are negative or 0!"
            STOP

         CASE("No match dim")
            WRITE(*,*) "ERROR: matrices dimensions mismatch:
   &          can not proceed with matrix product."
            STOP
```

```
      CASE("Timem<0")
         WRITE(*,*) "WARNING: time for manual operation is
  &          can be meaningless!"

      CASE("TimeT<0")
         WRITE(*,*) "WARNING: time for manual operation
  &          using tranposed left matrix can be meaningless!"

      CASE("TimeF<0")
         WRITE(*,*) "WARNING: time for automatic operation
  &          using matmul can be meaningless!"

      CASE DEFAULT
         ! Nothing to do
         WRITE(*,*) "WRONG ERROR IDENTIFIER: PLEASE,
   &         CHECK THE ERROR STRING"
         STOP

    END SELECT
  END IF

  RETURN

END SUBROUTINE Print_error
```

The second subroutine is more complicated: it works during execution, opening some files and printing to them the *temporary results* that the program is doing while running. It also print to a "global" file all steps for which it is called, enumerating calls and writing also the identifier it was called with.
Its arguments are:

- A *Debug* flag, stating the activation of the calls
- An *IO* string, which should be equal to "Read" or "Write" and tells if the call to the subroutine has to read the proper file or to write on it (if string is different from this, the subroutine acts nothing).
- A *label* which is intended to be used to know what variable is under analisys and in which file temporary results are stored. File names are *".Status_"+label+".log"*. They start with a dot to be hidden files on Ubuntu, with the idea that one goes looking for them only if she really needs. (To enable the view of hidden files in Ubuntu, usually the *Ctrl-H* shortcut can be used in any directory).
- real_matrix, ... : these arguments are program specific and represents the variables that have to be printed or used to print files.

```
! Writing CHECKPOINT subroutine for REAL MATRICES
  SUBROUTINE Checkpoint(Debug, IO, label, real_matrix, dims)
  ! Subroutine used to save/retrieve stuff from file
  ! Arguments are:
```

```
! Debug   : logical, flag which confirms the call for debugging
! IO    : string, should be "Read" or "Write", tells if read or write on log file
! label   : string. File names are: ".Status_" + label + ".log"
! real_matrix : real*8 2D array (in this specific case); is a variable to be
   stored
! dims    : integer*8 array with 2 elements. Used to eventually print a 0s
   matrix in the file.
!         dims should correspond to actual real_matrix dimensions.


IMPLICIT NONE

LOGICAL, INTENT(IN) :: Debug
CHARACTER(LEN=*), INTENT(IN) :: IO, label
CHARACTER(50) :: file_name
INTEGER*8, DIMENSION(2), INTENT(IN) :: dims
REAL*8, DIMENSION(dims(1),dims(2)), INTENT(INOUT) :: real_matrix
INTEGER*8 :: stat, ii, jj, kk, counter
COMMON counter

file_name = ".Status_" // TRIM(label) // ".log"
file_name = TRIM(file_name)


IF (Debug) THEN
   ! Retrieving data
   IF (IO.EQ."Read") THEN

      ! Opening file to read previous state
      !  (iostat is a collecting error variable)
      OPEN(10, file=file_name, status='old',
      &            action='READ', iostat=stat)

      IF (stat.EQ.0) THEN
         READ(10,*) real_matrix
      ELSE
         ! If no data to retrieve, then start from scratch
         DO ii=1,dims(1)
            DO jj=1,dims(2)
               real_matrix(ii,jj)=0d0
            END DO
         END DO
      END IF

      ! Closing file
      CLOSE(10)

   END IF


   ! Writing/saving data
```

```
     IF (IO.EQ."Write") THEN

        ! Opening file to write current state
        OPEN(10, file=file_name, status="REPLACE",
        &             action="WRITE", iostat=stat)

        ! Writing data to file
        WRITE(10,*) real_matrix

        ! Closing file
        CLOSE(10)


        ! Opening another file in which save whole history

        ! Creating file if not existing
        IF (counter.EQ.0) THEN
           OPEN(10, file=".debug_history.log", status="REPLACE")
           CLOSE(10)
        END IF

        OPEN(10, file=".debug_history.log", status="old",
        &          action="WRITE", iostat=stat, position="append")
        WRITE(10,*) ""
        WRITE(10,*) "Call n: ", counter, label
        counter = counter + 1
        WRITE(10,*) ""
        WRITE(10,*) real_matrix

        CLOSE(10)


     END IF
  END IF

END SUBROUTINE Checkpoint
```

The code above was written in a file called *Dubug.f*; such file is included in the code where the "main program" is.
In the modified program, at beginning some tests are done on dimensions to check if they are greater than 0 or equal to 0; also, another test is done to check that the matrix product can be done. Immediately after these tests the *Print_error* subroutine is called to print proper error messages.
In the following piece of code, the calcoulos are executed by some loops: here the *Checkpoint* subroutine is called before and after every step to eventually read from file the "previous interrupted" elaboration and overwrite it with the new one (the code is not copied here due to its length; see file "Ex3_Tommaso_Tabarelli_CODE.f").

Now, some notes on file compilation and scripting:

- Using "standard" compiler some "indentation" issues arose: to avoid them one can use a "modern" fortran compiler or pay more attention on fortran indentation and syntax rules. For exam-

ple, in the imported file the whole code must be indented, othewise the compiler recognize the 2 files as 2 different programs and it rises errors.

- As one can undestand, the checkpoint method is not acting as wanted: it is only a subroutine that overwrites some files, but it does not stop the program to a certain point or at every step (for example, for 1 or 2 seconds, to be able to check manually the ongoing process updating files while running). For this reason a complete "history" of the checkpoints is also created.

- A problem concerning *log* files is that they should be erased at a certain point otherwise computations will be seen as *already performed* every time the program is run with same *labels*, but it is not clear when exactly act this. The choice was to clean them every time a new compilation is done. For this reason, 2 bash scripts have been create: one to compile and creating an executable; a second one, called by the first, which delete all files in the current folder whose name start with "*.Status_*" (there can be some bash warning about the fact that the files do not exist, but they are not ruining compilation) and delete the ".debug_history.log" file. To compile, open terminal in the same folder and type "*source compile*". Then, to run the executable type "*source run*".

Results

The program runs well, the files and the output are as expected. Some tries were made using negative dimensions, 0 dimensions, dimensions not compatible with matrix product: the program worked as expected, printing the error and stopping the execution.

Self-evaluation

During this exercise I learned what debug and checkpoints are, how to implement them in a simplified way, how to define subroutines, how to include other fortran files and how to do it properly.
In my opinion, some pieces of the program can be examinated more deeply and andjusted properly. For example, my checkpoint routine is not very scalable since it creates a lot of files and they may be large due to the data stored in them: a way to fix this can be save only a fixed number of steps, but in that case it would be hard to manage with a language as fortran. Another thing that can be done is to ask for matrices dimensions as inputs, which is already prepared in the program (it is markdown before matrix dimension assignment).
Other things that can be done are checks for variable type, which in fortran are complicated since there is no built-in function doing this. To check them, a way can be to define more functions with different arguments type and to use an interface to call them; in this way the program take care of calling the proper function and they can print a unique message to be identified runtime, thus identifying also the type of variable.