Quantum information and Computing
Ex. 03

Andrea Nicolai - 1233407

October 26, 2020

Abstract

In this work we implemented an efficient system for debugging our code: some basic Test Units were written and run. They were exploited in some pre and post conditions, whose role was to check whether some properties of linear algebra were satisfied, as well as some variables were meaningful. In addition, we implemented the possibility to either just debug or print in output the variables and the checks being done.

Theory

Preparing the code for an easy debugging might seem tedious, but plays a fundamental role. Therefore we were asked to implement a function that might help pursuing this purpose. In order to do so, we exploited what software engineers call "Unit-Testing": its goal is to isolate each part of the program, and show that the individual parts are correct and behave as expected.
Moreover we were asked to write a short documentation, present in the header of the source file, and some pre and post conditions. These are some useful checks done respectively before and after a certain process, to be sure that everything behaved as expected.

Code Development

A module was implemented, called *CHECKPOINT_DEBUG_MODULE* for the purpose of this work. Our idea was to introduce two levels of debugging in a program that tests the rapidity of different definitions for the matricial product. Here, we will discuss only the part of the code related to debugging, since the remaining part had already been presented in a previous homework. The first level was mainly for testing and returns a simple print statement whether this test was successful. Whereas the second one involved printing in output both variables being tested and the test taking place. Another flag was introduced to exit the code with error(2) in the case the check fails, so to not run the entire program and save resources. This behavior was obtained by three flags, that were respectively:

```
LOGICAL :: DEBUG
LOGICAL :: DEBUG_PRINT_OUTPUT
LOGICAL :: EXIT_ON_ERROR
```

We need now to discuss what were the tests we implemented. We took into account only few types that may be helpful for our homework:

```
INTEGER*4,
INTEGER*8
LOGICAL
CHARACTER(*)
```

and all operators that can be defined according to those:

```
INTERFACE assert_eq
    PROCEDURE integer4_assert_eq, integer8_assert_eq, &
    & logical_assert_eq, string_assert_eq
END INTERFACE

INTERFACE assert_neq (...)
INTERFACE assert_greater (...)
INTERFACE assert_greater_equal (...)
INTERFACE assert_less (...)
INTERFACE assert_less_equal (...)
INTERFACE assert_positive (...)
INTERFACE assert_negative (...)
```

The following code is a prototype for one of the procedures of the interface *INTERFACE assert_eq* that implements a "check" whether two variables are equal. In particular, this takes in input two *INTEGER*4* and returns a bool that states whether this check has been successful. If the test fails, and the *exit_error* is *TRUE*, then exit the whole program without running it entirely.

```
FUNCTION integer4_assert_eq(arg1, arg2, output_flag, exit_error) &
& RESULT(passed)
    IMPLICIT NONE
    INTEGER*4, INTENT(IN) :: arg1, arg2
    LOGICAL, INTENT(IN)   :: output_flag
    LOGICAL :: passed, exit_error
    CHARACTER(32) :: arg1_str, arg2_str

    passed = (arg1.EQ.arg2)
    IF (output_flag) THEN
        WRITE(arg1_str, '(I11)') arg1
        WRITE(arg2_str, '(I11)') arg2
        CALL OUTPUT_SUBROUTINE(TRIM(arg1_str), TRIM(arg2_str) &
        & , "==␣", passed)
    END IF
    IF (exit_error.AND.(passed.EQV..FALSE.)) THEN
         CALL EXIT(2)
      END IF
END FUNCTION
```

Note as in the whole code procedures of this kind were repeated for all interfaces and for all types that might fit the operator we are implementing. That is to say that two strings can be tested to see whether they are equal, but would be meaningless to check whether one is "less or equal" than the other one. However, for a couple of $INTEGER * 4$ or $INTEGER * 8$ these operators can be still applied. In addition, an attempt was made to create a polymorphic function (or better a subroutine, as requested in assignment) taking as input two objects $arg1, arg2$ of the generic $CLASS(*)$ and, based on their type, making an explicit call to the proper procedure matching those types by using a $SELECT\ TYPE(arg)$. However, the latter is able to check only one object per time, while we needed a check on both of them simultaneously. Some workarounds for this issue might have been implemented in standards more recent than $FORTARN90$, but since we want to use this standard, we had to give up because this path would lead to nowhere.
Recalling the debug flags we introduced before, in the case we wanted to print the variables passed as arguments and the operator used for the check (e.g. "==" for equal, "<=" for less and equal and so forth), we set as $.TRUE.$ the $DEBUG\_PRINT\_OUTPUT$ in the main program. If the assertion (for example "the two variables are equal") is not verified, then it is printed an output that the check failed.
The following code is a subroutine that, when called in a procedure only if the $DEBUG\_PRINT\_OUTPUT$ is $.TRUE.$, prints the two variables, once they have been converted to strings, and the test being done. Finally, it asserts whether test has been successful.

```
SUBROUTINE OUTPUT_SUBROUTINE(var1_str, var2_str, &
                            & operat, test_passed)

   CHARACTER(*), INTENT(IN) :: var1_str, var2_str
   CHARACTER(*), INTENT(IN) :: operat
   LOGICAL, INTENT(IN) :: test_passed
   CHARACTER(1) :: NEW_LINE_CHAR = char(10)

   PRINT*, NEW_LINE_CHAR, "--------------------", &
   & NEW_LINE_CHAR, TRIM(ADJUSTL(var1_str)),"␣␣" , &
   & operat, TRIM(ADJUSTL(var2_str)),"->", test_passed
IF(test_passed.neqv..TRUE.) THEN
   PRINT*, "CHECK␣FAILED"
END IF
(...)
END SUBROUTINE
```

Since we wanted to efficiently store matrices and do tests, we defined a new derived type $MATRIX$ as we did in the last exercises. It was implemented in the $MATRIX\_UTIL$ module, specially modified and written for this report:

```
TYPE MATRIX
!Matrix dimension (rows, columns)
    INTEGER, DIMENSION (2) :: dims
    REAL*4, DIMENSION(:, :), ALLOCATABLE  :: element
END TYPE MATRIX
```

Thus we were able to initialize a matrix using a single function and call the proper functions for some of its objects when testing.
Our program is about multiplying matrices, thus we first need that dimensions of the matrix instantiated must be positive, and moreover them to be square. Therefore, as preconditions we introduced:

```
!PRE-CONDITIONS step1
IF (DEBUG) THEN
   PRINT*, "Check␣if␣matrix␣size␣variable␣is␣positive"
   PRINT*, assert_positive(size_matrix, DEBUG_PRINT_OUTPUT, &
   & EXIT_ON_ERROR)
END IF
```

Then, the second one:

```
!PRE-CONDITIONS step2
IF (DEBUG) THEN
   PRINT*, "Check␣matrices␣are␣square"
   PRINT*, "Matrix␣A␣->", assert_eq(matrixA%dims(1), &
   & matrixA%dims(2), DEBUG_PRINT_OUTPUT, EXIT_ON_ERROR)
 (...)
END IF
```

When product has been done, we check whether dimension of the *matrixC*, namely the resulting matrix, has meaningful dimensions wrt to the input ones:

```
!POST-CONDITIONS step
IF (DEBUG) THEN
   PRINT*, "Check␣whether␣resulting␣matrix␣is␣square"
   PRINT*, assert_eq(matrixC%dims(1), &
   & matrixC%dims(2), DEBUG_PRINT_OUTPUT)
   PRINT*, "Check␣whether␣resulting␣matrix␣&
␣␣␣&␣has␣same␣dims␣as␣input␣matrices"
   PRINT*, assert_eq(matrixC%dims(1), &
   & matrixA%dims(1), DEBUG_PRINT_OUTPUT, EXIT_ON_ERROR), &
   & assert_eq(matrixC%dims(2), &
   & matrixB%dims(2), DEBUG_PRINT_OUTPUT, EXIT_ON_ERROR)
END IF
```

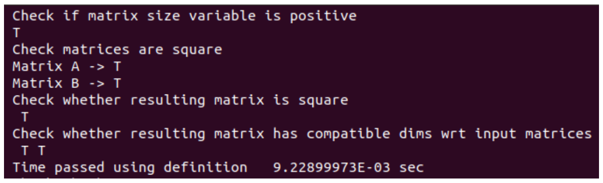
These last assertions were actually tested for all the three different matricial products, namely the one where index denoting rows runs faster, the one where the column related index runs faster, and finally the intrinsic $FORTRAN90$ function.

Results

Using a standard matrix size for the square matrix, i.e. 100, with the choice of flags:

```fortran
LOGICAL :: DEBUG = .TRUE.
LOGICAL :: DEBUG_PRINT_OUTPUT = .FALSE.
```

we obtain the following output, that is cut right after the first matricial product for a matter of space:



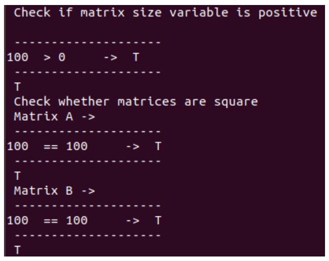
```
Check if matrix size variable is positive
T
Check matrices are square
Matrix A -> T
Matrix B -> T
Check whether resulting matrix is square
 T
Check whether resulting matrix has compatible dims wrt input matrices
 T T
Time passed using definition   9.22899973E-03 sec
```

Figure 1: Terminal output obtained when running the program having set the debug flag as $TRUE$ and the output debug flag as $FALSE$.

On the other hand, setting both them as $.TRUE.$ as:

```fortran
LOGICAL :: DEBUG = .TRUE.
LOGICAL :: DEBUG_PRINT_OUTPUT = .TRUE.
```

we obtain an output that looks like:



```
 Check if matrix size variable is positive

 --------------------
100  > 0     ->  T
 --------------------
 T
 Check whether matrices are square
 Matrix A ->
 --------------------
100  == 100     ->  T
 --------------------
 T
 Matrix B ->
 --------------------
100  == 100     ->  T
 --------------------
 T
```

Figure 2: Terminal output obtained when running the program having set the debug flag as $TRUE$ and the ouput debug flag as $TRUE$.

Here instead, we cut the output after the few checks for a matter of space. Note as each check is printed as output, both showing the variables and the operator applied to them. It is present the result of the test, too.

Self Evaluation

We have learned how to write a simple, but very time consuming, sort of debugging for our code. Some other operators might be be further included and, in addition, checks for other $FORTRAN90$ native types , such for example $REAL$ numbers, and derived types introduced by the user. This would require more work since, as an example, we cannot state that two numbers are equal unless we before have defined a precision. Therefore test would need to be rewritten from scratch. Nonetheless this approach may turn out to be very flexible and useful for further works.