# The *debugging* practise

**Abstract**

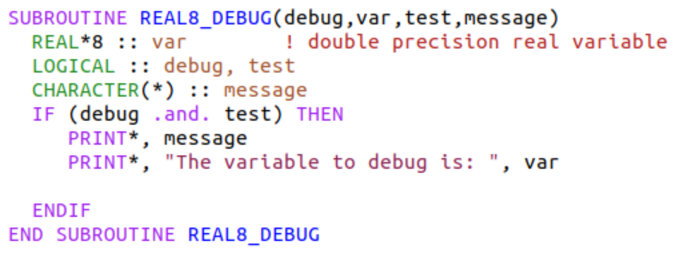
This exercise requires two tasks. The first one is writing a subroutine to debug a code. The latter can be useful also in future works. The second one is adding some useful practises, such as checkpoints, documentation and conditions, to the exercise three of the first assignment. The aforementioned exercise required to implement the matrix matrix multiplication in different ways, monitoring the performances of each method.

## Code development

Firstly a `MODULE`, named `DEBUGMOD`, is implemented. Its purpouse is to debug a code: therefore, to make it applicabile also to other codes, different `SUBROUTINE`s, that accept different type of variables, are created. These `SUBROUTINE`s are included into the `INTERFACE` called `DEBUG`. In particular, each `SUBROUTINE`, takes in input:

- Two logical flags: `debug` and `test`. They activate, respectively, the debug and the check that is performed in the program.
- A `CHARACTER`, that is the warning message that the user wants to print.
- The variable, `var`, over which we want to perform our debugging.

The figure 1 shows an example of this `SUBROUTINE`, for a double precision real variable.



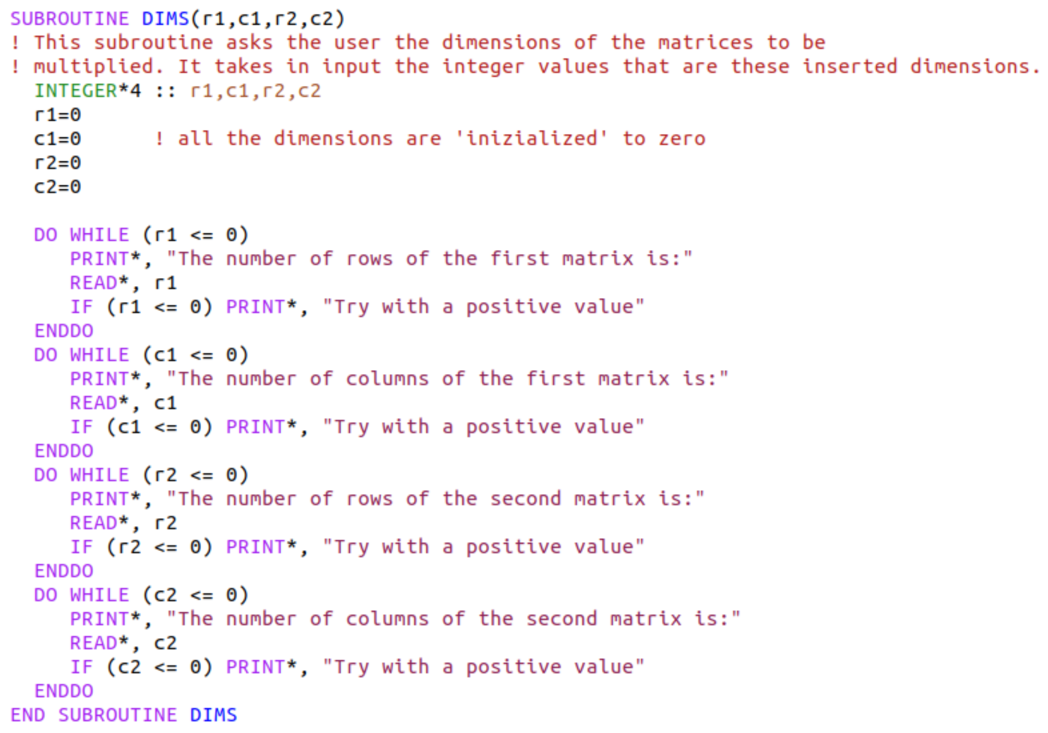
```fortran
SUBROUTINE REAL8_DEBUG(debug,var,test,message)
  REAL*8 :: var        ! double precision real variable
  LOGICAL :: debug, test
  CHARACTER(*) :: message
  IF (debug .and. test) THEN
     PRINT*, message
     PRINT*, "The variable to debug is: ", var

  ENDIF
END SUBROUTINE REAL8_DEBUG
```

Figure 1: An example of debug `SUBROUTINE`

Then, there is another `MODULE`, called `matrix`, similar to the one in the third exercise of the first assignment, that contains more documentation and comments

as requested. Therefore, to see the details on the construction of this MODULE, one can check the documentation on the attached code, without being verbose here.
I highlight that the MODULE contains a SUBROUTINE, DIMS, reported in figure 2, that asks the user to insert the dimensions of the matrices to be multiplied, with the *condition* that check if they are non negative. A DO WHILE loop repeats the request until the user inserts all positive dimensions.



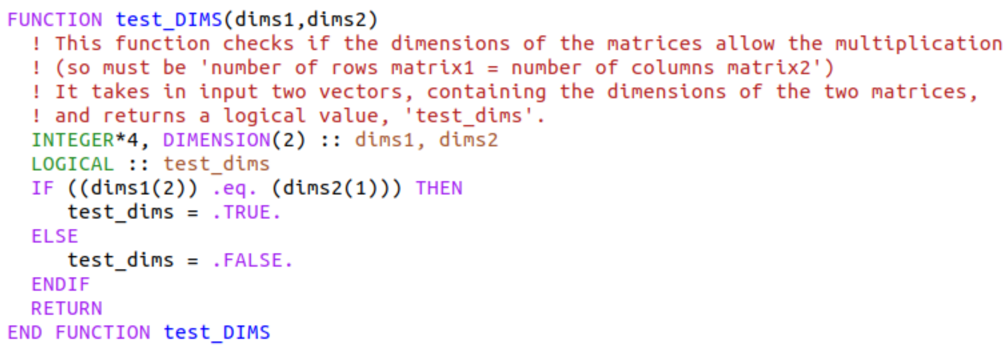
```
SUBROUTINE DIMS(r1,c1,r2,c2)
! This subroutine asks the user the dimensions of the matrices to be
! multiplied. It takes in input the integer values that are these inserted dimensions.
  INTEGER*4 :: r1,c1,r2,c2
  r1=0
  c1=0      ! all the dimensions are 'inizialized' to zero
  r2=0
  c2=0

  DO WHILE (r1 <= 0)
     PRINT*, "The number of rows of the first matrix is:"
     READ*, r1
     IF (r1 <= 0) PRINT*, "Try with a positive value"
  ENDDO
  DO WHILE (c1 <= 0)
     PRINT*, "The number of columns of the first matrix is:"
     READ*, c1
     IF (c1 <= 0) PRINT*, "Try with a positive value"
  ENDDO
  DO WHILE (r2 <= 0)
     PRINT*, "The number of rows of the second matrix is:"
     READ*, r2
     IF (r2 <= 0) PRINT*, "Try with a positive value"
  ENDDO
  DO WHILE (c2 <= 0)
     PRINT*, "The number of columns of the second matrix is:"
     READ*, c2
     IF (c2 <= 0) PRINT*, "Try with a positive value"
  ENDDO
END SUBROUTINE DIMS
```

Figure 2: 'DIMS' SUBROUTINE

Moreover a function, reported in figure 3 tests if the dimensions of the matrices allow the multiplication. Then this MODULE contains three functions, matrix_multiplication1, matrix_multiplication2 and matrix_multiplication3 that take in input two real double precision matrices and perform the multiplication, in three different ways (as requested in the first assignment).

The program called EX3 uses the aforementioned MODULEs and fill the matrices to be multiplied with random numbers, drawn from the uniform distribution in the interval [0,1]. Using the Fortran function CPUTIME, the time taken to do the matrices multiplication is printed in .txt files. The operation is done in four different ways: through the functions matrix_multiplication* (where $*$



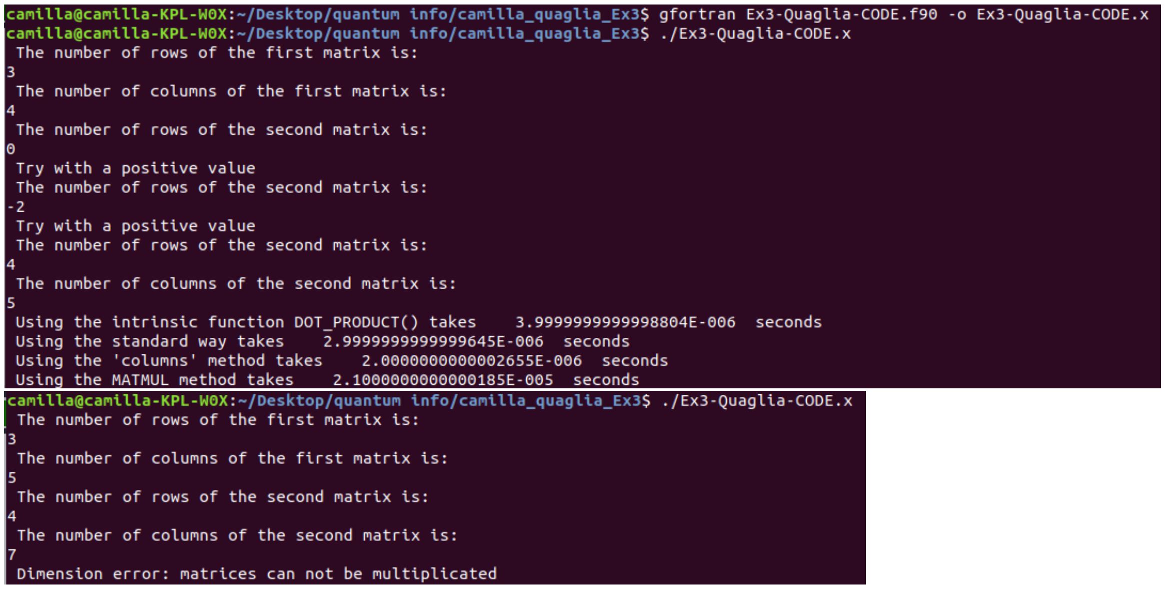
```fortran
FUNCTION test_DIMS(dims1,dims2)
  ! This function checks if the dimensions of the matrices allow the multiplication
  ! (so must be 'number of rows matrix1 = number of columns matrix2')
  ! It takes in input two vectors, containing the dimensions of the two matrices,
  ! and returns a logical value, 'test_dims'.
  INTEGER*4, DIMENSION(2) :: dims1, dims2
  LOGICAL :: test_dims
  IF ((dims1(2)) .eq. (dims2(1))) THEN
     test_dims = .TRUE.
  ELSE
     test_dims = .FALSE.
  ENDIF
  RETURN
END FUNCTION test_DIMS
```

Figure 3: 'test_DIMS' FUNCTION

stands for 1,2,3) and the Fortran intrinsic function `MATMUL`. The logarithm of the time taken for the operation vs the size (i.e. number of entries) of the resulting matrix is plotted, through a `GNUPLOT` script, called `plots.gnu`. The results are shown in the next section.

## Results

When we run the program we get an output of the tipe in figure 4



```
camilla@camilla-KPL-W0X:~/Desktop/quantum info/camilla_quaglia_Ex3$ gfortran Ex3-Quaglia-CODE.f90 -o Ex3-Quaglia-CODE.x
camilla@camilla-KPL-W0X:~/Desktop/quantum info/camilla_quaglia_Ex3$ ./Ex3-Quaglia-CODE.x
 The number of rows of the first matrix is:
3
 The number of columns of the first matrix is:
4
 The number of rows of the second matrix is:
0
 Try with a positive value
 The number of rows of the second matrix is:
-2
 Try with a positive value
 The number of rows of the second matrix is:
4
 The number of columns of the second matrix is:
5
 Using the intrinsic function DOT_PRODUCT() takes    3.9999999999998804E-006  seconds
 Using the standard way takes    2.9999999999999645E-006  seconds
 Using the 'columns' method takes    2.0000000000002655E-006  seconds
 Using the MATMUL method takes    2.1000000000000185E-005  seconds
camilla@camilla-KPL-W0X:~/Desktop/quantum info/camilla_quaglia_Ex3$ ./Ex3-Quaglia-CODE.x
 The number of rows of the first matrix is:
3
 The number of columns of the first matrix is:
5
 The number of rows of the second matrix is:
4
 The number of columns of the second matrix is:
7
 Dimension error: matrices can not be multiplicated
```

Figure 4: An example of the output

The user inserts the dimensions and if they are negative or don't allow the multiplication, a warning message is printed. Moreover we can see also the time, in seconds, taken for each multiplication method.

The plot described in the end of the previous section is showed in figure 5. For each multiplication method, we have four points, corresponding to the size of the resulting matrix of 20,300,10000 and 90000 entries. Generally, we can say that as the size increases the time taken increases. The better performance is obtained by the `MATMUL` intrinsic function, while the worst through the direct calculation 'rows times columns'.

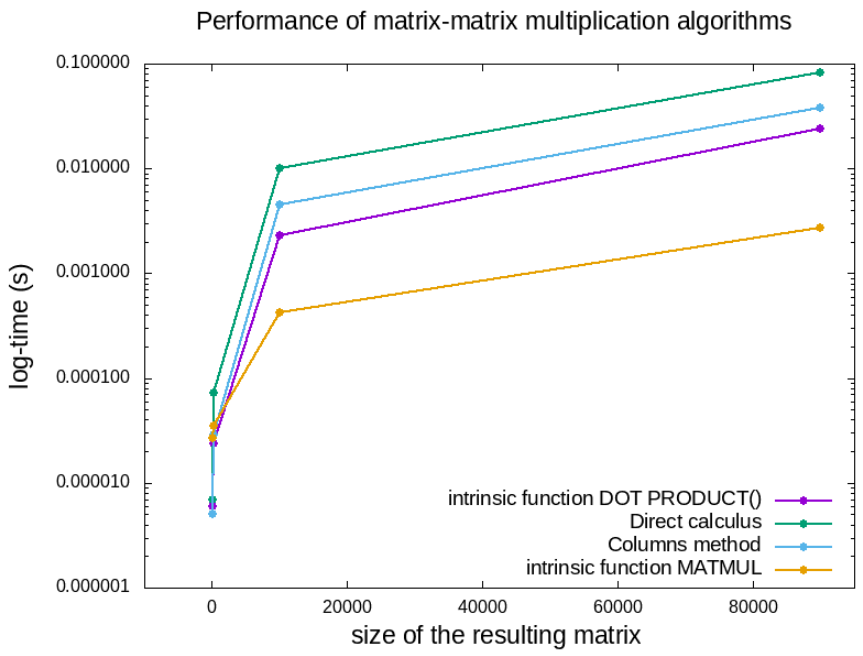


Figure 5: Plot of the logarithm of the time taken in the matrix-matrix multiplication vs the size of the resulting matrix, for four different multiplication methods.

## Self-Evaluation

The practise of adding documentation and comments is useful to make the code more readible by others and understandable for future works. Moreover the `'DEBUGMOD' MODULE` can be generalized to future works, so with this assignment one can become familiar with the concept of *flexibility*. Also insert checkpoints is a good habit to make the usage of a program easier.