Exercise 3: *Debuggers in FORTRAN 90/95.*

Iriarte Delfina (1231682)

Quantum Information

October 27, 2020

Abstract

In this report, a **debugging MODULE** is implemented in order to test any program and to induce a USER to **correct it errors** by supplying a MESSAGE in the bash. In particular, this will be performed in a program that computes the performances of different approaches by multiplying a $2D-$matrix. Several subroutines are provided in accordance with the utilities of this program in order to works as a debugger, generally, it corresponds in checks on the dimension of the matrix on purpose.

1 Introduction

A ***Debugger tool*** is a key component in any program since it allows to test errors and to help to determine the causes of it. GNU FORTRAN has various special options that are used for debugging such as the flag option `-ffpe-trap` and others. In this project, a **debugging MODULE** purpose is implemented. Inside of it, several subroutines can be found that help a USER to control the program in a smoother way. Moreover, good practice techniques to make the code simpler and avoid errors are shown.

Finally, the debugging MODULE is implemented in a main program that compares the performances of different approaches that carry out a matrix multiplication which, in general, is one of the **bottlenecks** in computing programming. The definition of matrix multiplication for an $m \times n$ and $n \times p$ matrices is simply given by a matrix $m \times p$ with entries:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1}$$

for $i = 1, \ldots m$ and $j = 1, \ldots, p$.

2 Code development

A good practice in FORTRAN 95 that may sound tedious is to use the `IMPLICIT NONE` statement since it prevents potential confusion in variable types and makes detection of typographic errors easier. It is also optimal to use a MODULE for precision purposes as shown in Listing 1. The implementation among the project is simply by TOKENIZING the precision that we would like, explicitly `use precision, pr=>dp`. Hence, any precision can be implemented and change it easily avoiding any kind of confusion.

```fortran
module precision
    implicit none
    integer, parameter :: sp = kind(1.0)
    integer, parameter :: dp = kind(1.d0)
    !integer, parameter :: qp = kind(1.q0)
end module precision
```

Listing 1. Module precision

Another important good programmer technique is to **document** the code with the main concepts so that it can be easily readable in the future. The main module of our project is called MODULE DEBUGGING and it contains the following subroutines:

- `check_debug(debug, msg, input)`: It takes as input a `logical` value, called DEBUG, that when is `.True.` it enables the debugging. Hence a MESSAGE, that is an `allocatable character`, is optionally display in the bash. In addition, an INPUT variable, that can be of any `class` and it is also optional, can be inserted and, thanks to the `select type` command, a MESSAGE with the correspondent type will appear. The implementation is shown in Listing 2.

```
subroutine check_debug(debug, msg, input)
      implicit none
       logical, intent(in) :: debug
       character(:), allocatable, optional :: msg
       class(*), intent (in), optional :: input

       if (debug.eqv..true.) then
           write(*,*) '-----------'
           write(*,*) msg
           if (present(input)) then
               write(*,*) 'type of the varible:'
               select type(input)
                   type is (integer(2))
                   write(*,*) "Got an integer of kind 2"
                   type is (integer(4))
                   write(*,*) "Got an integer of kind 4"
                   type is (integer(8))
                   write(*,*) "Got an integer of kind 8"
                   type is (real(4))
                   write(*,*) "Got a real of kind 4"
                   type is (real(8))
                   write(*,*) "Got a real of kind 8"
                   type is (logical)
                   write(*,*) "Got a logic"
                   type is (complex)
                   write(*,*) "Got a complex"
               end select
           end if
       end if
   end SUBROUTINE
```

Listing 2. Subroutine check_debug

- `check_dimension(debug, nrow, ncol)`: Takes as input the DEBUG flag previously defined and the number of rows and columns of a matrix. Again, enabling the DEBUG flag we allows the debugging to occur: we check if the inserted number of rows and columns are negative and the subroutine `check_debug` is call in order to display a message. If the variables are invalid, the idea is that instead of breaking the whole program, which is sort of scary, ask the USER to reconsider the parameters and insert new ones in the bash. This will be done repeatedly until the correct dimension of the parameters are given. Once everything is fine, a message saying 'ALL PERFECT' is display. The implementation is shown in Listing 3.

```
SUBROUTINE check_dimension(debug, nrow, ncol)
IMPLICIT NONE
   integer(dp), intent(inout) :: nrow, ncol
   logical, intent(in) :: debug
   character(:), allocatable :: msg
```

```
    msg = 'Checking if the dimensions are positive:'
    call check_debug(debug, msg)
    if (debug.eqv..true.) then
        do while (nrow <= 0 .or. ncol <= 0)
            write(*,*) 'Dimension of the matrix not valid. Insert again:'
            write(*,*) '------------------------------------------------'
            write(*,*) "Number of rows desire:"
            read(*,*) nrow
            write(*,*) "Number of cols desire:"
            read(*,*) ncol
        end do
        write(*,*) 'All perfect'
    end if
    END SUBROUTINE
```

Listing 3. Subroutine for checking invalid matrix dimension

- `product_versality(debug, nn, kk, ll, mm)`: As before, the DEBUG flag is taken as input and the `check_debug` is called and display a message if the DEBUG is enable. Furthermore, the subroutine check if the inner dimension of two matrices are valid, in order to compute the product among them, and if not the USER has to insert new values in the bash. The implementation is similar as before and it is shown in Listing 4.

```
subroutine product_versality( debug, nn, kk, ll, mm)
    implicit none
    integer(dp), intent(inout) ::  nn, ll, mm, kk
    integer(dp) ::inner
    logical, intent(in) :: debug
    character(:), allocatable :: msg
    msg = 'Checking if the matrices dimension for product purposes are ok:'
    call check_debug(debug, msg)
    if (debug.eqv..true.) then
        if (ll .ne. kk) then
            write(*,*) 'Do you really think that you can do a matrix product with this
    dimensions?'
            write(*,*) '------------------------------------------------'
            write(*,*) "Inner dimension of the matrices:"
            read(*,*) inner
            ll = inner
        end if
    write(*,*) 'All perfect'
    end if
    end subroutine
```

Listing 4. Subroutine for checking the viability of matrix product

- `check_square_matrix(debug, mat)`: takes as input the DEBUG flag and a matrix; and compute if the matrix is square or rectangular, as shown in Listing 5.

```
subroutine check_square_matrix(debug, mat)
implicit none
    real (dp), intent(in) :: mat(:,:)
    logical, intent(in) :: debug
    character(:), allocatable :: msg
    msg = 'Square or rectangular matrix'
    call check_debug(debug, msg)
    if (size(mat,dim=1)==size(mat,dim=2)) then
        write(*,*) "Matrix is square"
    else
        write(*,*) "Matrix is rectangular"
    end if
```

```
end subroutine
```

Listing 5. Checkpoint: Square Matrix

- `matrix_parameter(mtx)`: returns the most important attributes of a matrix such as the the number of rows and columns, kind and its elements.
- `print_matrix(mat)`: Prints the elements of a matrix.

The MODULE DEBUGGING is implemented in the program TEST_PERFORMANCE that compares the computational performances of different approaches when computing the matrix multiplication. This is done by using the function `cpu_time` that allows to measure the time between computation. In particular, two SUBROUTINES are created that looped over the indices of i and j (and j and i respectively) as in equation 1. The implementation of one of them is shown in Listing 6. Additionally this is compared using the predefined function `matmul` provided by FORTRAN. Finally, all the results where saved in an output file.

```
subroutine matxmat_1(mat_1, mat_2, nn, ll, mm, mat_p)
 use precision, dp=>dp
    implicit none
    integer(dp) :: i, j, k
    integer(dp), intent(in) ::  nn, ll, mm
    real (dp), intent(in)       :: mat_1(1:nn, 1:ll), mat_2(1:ll, 1:mm)
    real (dp), intent(out) :: mat_p(1:nn,1:mm)
    mat_p(i,j) = 0.0
    do i = 1, nn
        do j=1,mm
            do k=1,ll
                mat_p(i,j)=mat_p(i,j)+mat_1(i,k)*mat_2(k,j)
            end do
        end do
    end do
end subroutine
```

Listing 6. Matrix multiplication

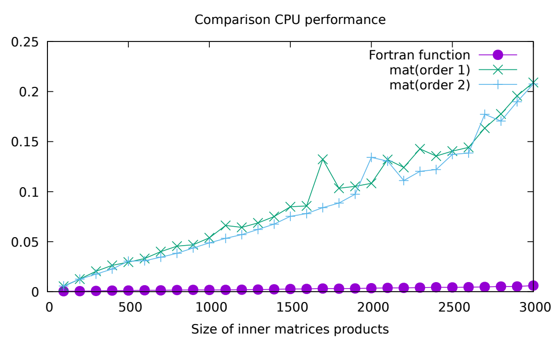
The checkpoints of the MODULE DEBUGGING in the TEST_PERFORMANCE are simply implemented by calling the correspondent subroutine as seen in Listing 7.

```
!First check: negative dimension.
write(*,*) 'Matrix 1 setup'
call check_dimension(debugg,n, l)
write(*,*) 'Matrix 2 setup'
call check_dimension(debugg, l, m)

!Second check: inner dimension of the matrices.
call product_versality(debugg,  n, l, l, m)

allocate(mat1(1:n,1:l), mat2(1:l, 1:m), mat_prod(1:n, 1:m), mat_result(1:n, 1:m), mat_result_2(1:n, 1:m))

call random_number(mat1)
call random_number(mat2)

!third check: parameters of the matrices
call matrix_parameter(mat1)
```

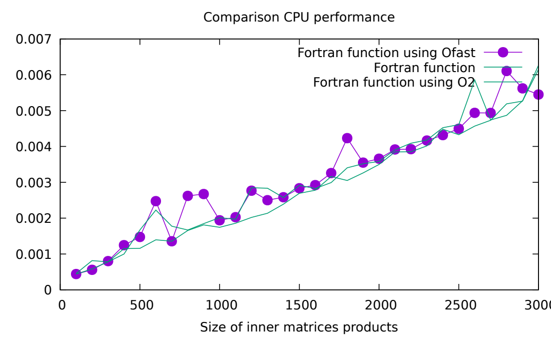
Listing 7. Matrix type

3 Results

The results obtained for the different approaches of matrix multiplication are shown in Figure 1. As it can be seen in Figure 1a, the matmul operation is much faster than the loop ones. In general, it can also be seen than implementing several optimization flags such as `-Ofast` or `-O2` does not really affect on the performance as shown in Figure 1b that correspond to the `matmul` function provided by FORTRAN.



(a) Performance of different matrix multiplication approaches with no optimization flag.



(b) Performance of the `matmul` function with several optimization flag. A similar behaviour is exhibit for the other approaches

Figure 1. Output results: Performances.

Regarding the debuggers, they turn out to be **useful** since it allows easy checks if different possible erroneous scenarios arises, such as wrong initialization of the matrices. The debugging is **optionally** for the USER and can be easily trigger by just setting the `.True.` option in the logical value. Using modules and subroutines allows a **simpler code** in the program and to easily **reuse** them in the future.

4 Conclusion

The implementation of a **DEBUGGER MODULE** in FORTAN was successfully performed and tested in a program that compares several approaches of matrix multiplication. It allows us to control in an easier way the errors that might occur and to prevent them.