UNIVERSITY OF PADUA
PHYSICS AND ASTRONOMY DEPARTMENT

Homework #3

Student name: *Beatrice Segalini* – 1234430

Course: *Quantum Information and Computing 2020* – Professor: *S. Montangero*
Due date: *October 26th, 2020*

Abstract

In this report, the description of a `Fortran90` program for executing matrix products is displayed. The code contains three different ways to perform the aforementioned computation. Other utility functions are also defined; moreover, documentation and debugging processes are also provided. These last features are fundamental for making the code more flexible and manageable also for different users and are the main focus of this exercise.

Theory

Given two matrices A and B, respectively of dimension $N \times M$ and $P \times Q$, the element $c(ii, jj)$ of the product matrix $C = A \times B$ is given by the following formula:

$$c(ii, jj) = \sum_{kk=1}^{M} a(ii, kk) \cdot b(kk, jj) \tag{1}$$

where $a(ii, kk)$ and $b(kk, jj)$ are the elements of the matrices A and B.
The product matrix C is defined if and only if:

$$M = P \tag{2}$$

and its size will be $N \times Q$.
When manually computing the matrix-matrix product, the *error* is defined as:

$$E = \sum_{ii=1}^{N} \sum_{jj=1}^{Q} |C(ii, jj) - C_F(ii, jj)| \tag{3}$$

where C_F represent the product matrix computed via the intrinsic `Fortran90` function.

Code development

Program structure. In this program, the matrix-matrix product is computed in three different ways (see Listing 1):

- with the intrinsic `Fortran90` function;
- with three nested cycles, with the indices in the following order: `kk`, `jj`, `ii`;
- with three nested cycles, with the indices in the following order: `ii`, `jj`, `kk`.

For each method, execution time is computed via `cpu_time` function.

Listing 1: Computation of matrix-matrix product.

```
! ###### Fortran intrinsic function ######
call cpu_time(start)
prodF = matmul(A, B)
call cpu_time(finish)

! ######## kk-jj-ii manual method ########
call cpu_time(start)

!       matrix multiplication with "manual" method:
!       first cycling on kk, then jj, then ii
do ii = 1, rowa
  do jj = 1, colb
    do kk = 1, rowb
      myprod1(ii, jj) = myprod1(ii, jj) + A(ii, kk)*B(kk, jj)
    end do
  end do
end do

call cpu_time(finish)

! ######## ii-jj-kk manual method ########
call cpu_time(start)

!       matrix multiplication with "manual" method:
!       first cycling on ii, then jj, then kk
do kk = 1, rowb
  do jj = 1, colb
    do ii = 1, rowa
      myprod2(ii, jj) = myprod2(ii, jj) + A(ii, kk)*B(kk, jj)
    end do
  end do
end do

call cpu_time(finish)
```

The matrices A, B considered in the computation are randomly generated via the user-defined function `RandMat`, included in the `matrix_operations` module (see full code for reference). The user is required to insert both matrices sizes and to decide whether or not to implement the debugging procedure (function `debugging` in module `debugging_mod`). Matrices are printed on screen via the `printmatrix` function of the `matrix_operations` module.

Finally, error is computed with the `ComputeError` function so as to evaluate the precision of the manually defined products, following the formula 3.

Documentation and Comments. In this exercise, one of the main goals is to familiarise with writing proper documentation for each program, module, function or subroutine that is used. The documentation is reported at the beginning of each module and program; an example is provided in Listing 2.

In addition, the main passages of the code are commented thoroughly in order to make the functionalities of the program more clear.

Usually, the aim of variables and a brief description of conditions and loops is included, making the code better organised and understandable by different users.

Listing 2: Example of documentation: full documentation of `matrix_operation` module.

```
! -----------------------------------------------------------------
! ----------------------- DOCUMENTATION ------------------------
! -----------------------------------------------------------------
! matrix_operations module:
!      contains utility functions for matrix handling
! -----------------------------------------------------------------
! RandMat:
!    function to initialize a N-by-M matrix (a) with random float
!    numbers in (0,10)
!   INPUT:
!       N = integer, number of matrix rows
!       M = integer, number of matrix columns
!   OUTPUT:
!       a = real*4, dimension(N,M), N-by-M matrix
! -----------------------------------------------------------------
! ComputeError:
!    function that compute the difference (element-wise) between two
!    matrices, then adds up the results in order to evaluate a certain
!    kind of error. Includes a check on matrix shapes.
!   INPUT:
!       mat1, mat2 = matrices of which we want to compute the "error"
!   OUTPUT:
!       e = real, defined as the sum of the element-wise differences
!           between @mat1 and @mat2
! -----------------------------------------------------------------
! printmatrix:
!    subroutine to print matrices in a clearer form
!   INPUT:
!       A = matrix to be printed
!   OUTPUT:
!       print on screen matrix A arranged in a "grid" form (elements
!       arranged in rows and columns)
! -----------------------------------------------------------------
! -----------------------------------------------------------------
! -----------------------------------------------------------------
```

Error handling and Checkpoints. The `debugging` function is defined in order to properly provide an easy way to check for errors and to simplify the debug process. The function checks if the `condition` (logical type, the only mandatory argument) is true: if that is the case, it means there is no issue detected. As a consequence, it produces an output on screen according to the other optional inputs, allowing the user to control the debugging process. The optional inputs are:

- `msg` = string, it is a text provided by the user for labelling the condition. It can be printed on screen;
- `verbose` = logical. If True, a very "long" version of the output is printed on screen, reporting the messages written by the user and the value of `content`, if present;
- `stopprg` = logical, if True the program is stopped at the first error detected;
- `content` = general variable on which the condition is verified, it can be printed with the `select type` construct (only a few cases are reported in 3). Of course, it is printed only if present and if we are selecting the `verbose` mode.

Three support variables are used in the subroutine in order to define the various options for the user (`verbose` mode, program stopping on errors, label message printing).

Listing 3: Code extract of the `debugging` function.

```
subroutine debugging(condition, msg, verbose, stopprg, content)
! support variables
full_text = (present(verbose).AND.verbose).OR.(.NOT.present(verbose))
stp = present(stopprg).AND.stopprg
msg_yes = present(msg)

if (msg_yes .AND. full_text) then
! "full" output mode
  if (condition .EQV. .TRUE.) then
    if (present(content)) then
! if variable is present, print it on screen
      select type(content)
        type is (integer(8))
          print*, msg, " => [OK], Variable = ", content
        type is (real(4))
          print*, msg, " => [OK], Variable = ", content
      end select
    else
      print*, msg, " => [OK]"
    end if
  elseif (stp) then
! exit program if stop condition is True
    print*, msg, " => [ERROR], Abort execution"
    stop
  else
! just print ERROR string if an error is detected but stp is False
    print*, msg, " => [ERROR]"
  end if
elseif (msg_yes) then
! less verbose output, just with msg print
  if (condition .EQV. .TRUE.) then
    print*, msg, " => [OK]"
  elseif (stp) then
    print*, msg, " => [ERROR], Abort execution"
    stop
  else
    print*, msg, " => [ERROR]"
  end if
else
! short output
  if (condition .EQV. .TRUE.) then
    print*, "[OK]"
  elseif (stp) then
    stop
  else
    print*, "[ERROR]"
  end if
end if
end subroutine
```

Pre-conditions. The pre-conditions that needs to be verified for this exercise are:

- a check on user-inserted values, i.e. matrix sizes must be positive and the condition 2 must be satisfied;
- all matrices must be properly allocated;

The first point is automatically checked via the debug process described before, while the allocation of matrices is manual in this case. It can be further automatised using the same `debugging` function, but it was chosen not to do so in order to make the program more comprehensible.

Post-conditions. The post-conditions checked is the correctness of the manual computation of matrix product: this is done via the `ComputeError` function and gives proper result. Another post-condition is the checking of proper deallocation of matrices at the end of the program. However, since the allocation is done manually, also this passage is implemented in the same way. Thanks to the flexibility of the `debugging` function, also this condition can be checked automatically, but this is not implemented in this version of the code.

Results

The code compiles and works properly. It has been tested with randomly generated matrices (initialised with the `RandMat` function) of different sizes and shapes, although not very big. The various combination of input parameters in the `debugging` function have been tested, obtaining the expected output.

Self evaluation

The code can be improved in different ways, e.g.:

- Despite the proper error handling introduced via the `debugging` function, the program still crashes if an incorrect type of input is inserted by the user.
- The program could be further automatised and less user-dependent.
- A check on allocation of variables could be also implemented.
- Testing on bigger matrices and with different optimization flags can also be performed.

To conclude, this exercise allows to learn about debugging and error handling, which are skills broadly useful to programmers. Furthermore, writing proper documentation and comments makes the code generally more readable and understandable for different users, to have complete insight about its functionalities.