Quantum Information and Computing 2020/21
Week 3 report

Rocco Ardino
(Dated: Monday 26th October, 2020)

In order to write user-friendly code, it is important to make it clear as much as possible and to implement debugging functionalities. A possibility is offered by the common practice of Fortran checkpoint subroutines. In this report, we implement a new module containing a simplified version of this subroutine and we apply the new functionality to a program where matrix matrix multiplications are performed. Moreover, pre- and post- conditions are attached to the operations to make debugging phase cleaner and more feasible. Lastly, we add a concise and complete documentation to the code written for the report.

1 Theory

Let us consider two matrices $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{p\times q}$, of dimensions respectively $n \times m$ and $p \times q$. If $m = p$, it is possible to compute the matrix product AB and the result is a matrix $C \in \mathbb{R}^{n\times q}$ of dimensions $n \times q$, with matrix elements:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \qquad \begin{matrix} i=1,\dots,n \\ j=1,\dots,q \end{matrix} \quad . \tag{1}$$

This algorithm is implemented in a very simple way through:

- a loop over k for the calculation of a single matrix element (complexity: $\mathcal{O}(m = p)$);
- two loops over i and j for the calculation of all the entries of the product matrix (complexity: $\mathcal{O}(nq)$)

Therefore, the overall complexity of the operation is $\mathcal{O}(nmq)$, being $\mathcal{O}(n^3)$ for the special case of two square matrices multiplication.

From a mathematical point of view, the order of the three loops is irrelevant for the result. However, it affects the performances depending on the way a language stores variables in memory. This study was done in the report for the week 1, so it is not treated here since we will focus on debug tools.

2 Code Development

The new module **`err_handling_mod`** is implemented in order to deal with the debugging of Fortran code. Its core component is the subroutine **`check`**, whose features are explained in the following Subsection 2.1. The implementation of other useful functions inside the module is described in Subsections 2.2 and 2.3.

2.1 Check subroutine

This subroutine has several input arguments to deal with multiple cases:

- **`debug`**: logical that enables/disables debug output messages;
- **`condition`**: logical representing the condition to verify in order to print debugging messages and/or stop the execution at checkpoint;

- **msg_type**: string representing the type of debugging message (for example, "Error" or "Warning");
- **msg**: string containing the debugging message;
- **trg_stop** (optional): logical that stops the execution, if enabled, when the condition argument is true;
- **var** (optional): generic variable to print if given.

The first operation is a check on the condition: if it is met, the subroutine enters in a block where another check on the debug flag is met. If also this check is satisfied, the subroutine enters in a block containing the logic for printing debug statements. The implementation of this part of code is sketched in Listings 1 and 2, respectively for message and variable printing. For Listing 1, note how the debugging message is attached to a string flag indicating the type of the message with a different color. For Listing 2, observe how the code is capable of discerning the type of the variable to print through a case selection block.

```
! choose among message type (error/warning) if given
if (msg_type == "Error") then
    print *, "["//achar(27)//"[1;31m"//msg_type//achar(27)//"[0m]: "//msg
else if (msg_type == "Warning") then
    print *, "["//achar(27)//"[1;95m"//msg_type//achar(27)//"[0m]: "//msg
else if (msg_type == "Debug") then
    continue ! do nothing
else
    print *, "["//achar(27)//"[1;32m"//msg_type//achar(27)//"[0m]: "//msg
end if
```

Listing 1. Implementation of message printing along with message type. Color and style from bash scripting are employed.

```
! print variable if var is given
if (present(var)) then
    ! switch over several cases for the type of the variable
    select type(var)
        type is (logical)
            print *, var, "logical variable"
        type is (integer(2))
            print *, var, "integer(2) variable"
        type is (integer(4))
            print *, var, "integer(4) variable"
        type is (real(4))
            print *, var, "real(4) variable"
        type is (real(8))
            print *, var, "real(8) variable"
        type is (complex(4))
            print *, var, "complex(8) variable"
        type is (complex(8))
            print *, var, "complex(16) variable"
        type is (character(*))
            print *, var, "character(*) variable"
    end select
end if
```

Listing 2. Implementation of variable printing through case select block.

When the subroutine exits the debug block, another condition block is met, namely the one involving the optional argument `trg_stop`. If enabled, it stops the execution of the program at checkpoint when the condition is met. Note that the stopping flag has a true default value and this

is possible through the declaration of a dummy variable `trg_stop_`. This trick and the condition block are showed in Listing 3.

```
! dummy arguments
logical :: trg_stop_

if (.NOT.present(trg_stop)) then
    trg_stop_ = .TRUE.
else
    trg_stop_ = trg_stop
end if

! check if error/warning/... condition is satisfied
if (condition) then

    if (debug) then
        ...
    end if

    ! stop execution if flag is enabled
    if (trg_stop_) stop

! end if block for condition
end if
```

Listing 3. Implementation of the stopping block along with default value.

Now, let us take into account a simple example of matrix matrix multiplication in order to explain a use case of checkpoint. Before applying the multiplication algorithm, one would check if the two matrices have the correct dimensions. Therefore, one could exploit a sketch of code similar to the one in Listing 4 as pre-condition.

```
N = size(mat1, 1)
M = size(mat1, 2)
P = size(mat2, 1)
Q = size(mat2, 2)

! pre-condition: check if mat2 can multiply mat1 (mat1.cols=?=mat2.rows)
call check(debug     = .TRUE., &
           condition = M.NE.P, &
           msg_type  = "Error", &
           msg       = "Cannot do mat1*mat2: size(mat1,2) =/= size(mat2,1)", &
           trg_stop  = .TRUE.)
```

Listing 4. Example of application of check subroutine to matrix matrix multiplication.

2.2 Dimensions check function

The function **`dim_check(mat1,mat2)`** is implemented in order to check if the dimensions of the input matrices `mat1` and `mat2` are the same. This information is stored in the result logical `AreDimEq`, as can be seen in the sketch of implementation in Listing 5.

```
! mat1 and mat2 dimensions
N1 = size(mat1, 1)
M1 = size(mat1, 2)
N2 = size(mat2, 1)
M2 = size(mat2, 2)

if ((N1.EQ.N2).AND.(M1.EQ.M2)) then
    AreDimEq = .TRUE.
else
```

```
    AreDimEq = .FALSE.
end if
```

Listing 5. Sketch of the implementation of **dim_check(mat1,mat2)** function for dimension check.

2.3 Matrix equality check function

The function **eq_check(mat1,mat2,thld)** is implemented in order to check if the input matrices `mat1` and `mat2` are the same at a precision level encoded in the optional threshold argument `thld`. Given A and B the two input matrices, the equality rule applied is the following:

$$|a_{ij} - b_{ij}| < \text{thld} \cdot \frac{1}{2}|a_{ij} + b_{ij}|, \qquad \forall i, j \quad . \tag{2}$$

Concerning the threshold argument, its default value is set to 10^{-6} with the same trick used for `trg_stop` in the checkpoint subroutine. The implementation of this function is sketched in Listing 6, where we can see that the equality information is returned in the logical `AreMatEq`.

```
! mat1 dimensions
N1 = size(mat1, 1)
M1 = size(mat1, 2)

AreMatEq = .TRUE.
! check if dimensions of mat1 and mat2 are the same
! otherwise return false
if (dim_check(mat1,mat2)) then
    ! loop over all matrix elements
    do jj=1,M1
        do ii=1,N1
            ! check if equality holds at a certain precision depending on thld
            ! otherwise exit the loop and return false
            if (ABS(mat1(ii,jj) - mat2(ii,jj)) &
                < 0.5*thld_*ABS(mat1(ii,jj) + mat2(ii,jj))) then
                continue
            else
                AreMatEq = .FALSE.
                exit
            end if
        end do
    end do
else
    AreMatEq = .FALSE.
end if
```

Listing 6. Sketch of implementation of **eq_check(mat1,mat2,thld)** function for equality check between matrices.

3 Results

The new module is tested on the last exercise of week 1. Pre- and post- conditions are now added respectively before and after matrix matrix multiplication. In fact, we want to be sure that, before reaching the matrix multiplication algorithm, the two input matrices have valid (non-negative) dimensions and can be multiplied. Moreover, we want to know if the results obtained for the same input but with different implementations of the algorithm, are the same at a certain precision level.

Note that the debugging functionalities are enable if the option `-debug` is given at command line to the executable. Moreover, the source code file is compiled by adding the flags `-Wall` and

`-ffree-line-length-0` in order to print compile warnings and avoid errors for too long lines. Then, the functionalities of the new module are tested by analyzing several cases, such as input matrices with:

- negative dimensions;
- size such that the multiplication is not possible;
- too large number of entries.

In each of them, the new checkpoint subroutine works with great flexibility and efficacy. Moreover, the other two functions of the new module confirm that the results of the matrix matrix multiplication algorithms, implemented during week 1, are identical with a threshold value of $\sim 10^{-14}$ for 1000×1000 input matrices.

4 Self-evaluation

The implementation of the code has led to successful results. A new module is now available for error handling. In particular, a flexible implementation of a checkpoint function can be used for future implementation of algorithms and to write new code in an easier and faster way. Moreover, adding more comments to the code and a concise documentation has made it easier to control and to review. On the other side, a more general way of handling errors through new user-defined types can be implemented for future developments of more complex code.