Week 3: Error Handling and Debugging

Alice Pagano

(Dated: October 26, 2020)

In order to write user-defined functions in a proper way and a code which return correct result in general, one should include several control conditions to check if the operations are performed correctly: pre and post condition and checkpoints must be added to the code for debugging. In this Report, we define new modules for debugging the code and handling errors. We slightly modify the module of the last week, in which the new matrix type was added, by integrating the new modules. In particular, we implement a function for matrix-matrix multiplication with all the required checks.

I. THEORY

In software development, **debugging** is the process of finding and resolving problems that prevent correct operation in the code. Generally, in programming languages such as Fortran, bugs may cause silent problems and it is often difficult to see where the initial problem happened. In those cases, specific tools may be needed. However, as a first approach, the debugging option provided by the compiler can be used. For instance, in the GNU compiler some useful option for debugging are:

- **`-Wall`**: to enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (such as unused variables);
- **`-fbacktrace`**: to specify that, when a runtime error is encountered or a deadly signal is emitted (segmentation fault, illegal instruction, bus error or floating-point exception), the Fortran runtime library should output a backtrace of the error;
- **`-ffpe-trap=list`**: to specify a list of floating point exception traps to enable such as `invalid` (invalid floating point operation, such as $\sqrt{-1}$), `zero` (division by zero), `overflow` (overflow in a floating point operation), `underflow` (underflow in a floating point operation) and so on;
- **`-fcheck=all`**: to enable run-time tests, such as, for instance, array bounds checks.

However, one can also implement its own code for debugging and in this Report we will show a very simple example of a debug module.

Errors in the code can be detected by the debugger or can arise during the execution of a program; **error handling** is the process of responding to the occurrence of exceptions during this normal flow which is in general interrupted by them. Fortran lacks a framework for error handling and in absence of an error handling mechanism built into the Fortran standard, one has to rely on self built modules. When developing a new framework for error handling, it should be developed in an evolutionary sense: existing code should require only minimal changes to be able to work with the framework. It should be possible to gradually introduce more advanced error handling features without significant code overhead, especially avoiding explicit initialisation [1]. In this Report, we start to develop a very simple error handling module. It includes operations which aim to be as general as possible, in order to maybe extend this module to a more complex one in a near future.

II. CODE DEVELOPMENT

A. Error Handling Modules

Firstly, we develop a simple framework to handle errors which aims to be as general as possible. It is composed by two main modules:

- `MODULE` **`ERROR_HANDLING`**: it handle general errors and contains subroutines which stops the program in the presence of an error and prints its info;
- `MODULE` **`ERROR_HANDLING_COMMON_ERRORS`**: it contains the definition of common errors which may arise.

In particular, the MODULE **ERROR_HANDLING** contains:

- a new **ERROR** type definition which includes two strings: the `info` string contains the information about the error while `method` stores in which function/subroutine the error occurs;

```
type, public :: error
    character(:), allocatable :: info
    character(:), allocatable :: method
end type error
```

- SUBROUTINE **create_error**(ifail,info,method): it takes as input an error instance `ifail`, its info and method strings and it calls the SUBROUTINE **error_constructor**. In this version of the code, this subroutine is not necessary and it is only a step more which add complexity to the code. However, if a more complex version of the error handling is implemented, this function will be necessary;

```
SUBROUTINE create_error(ifail,info,method)
    type(error), intent(out), optional :: ifail
    character(len=*), intent(in), optional :: info
    character(len=*), intent(in), optional :: method

    call error_constructor(ifail,info,method)
END SUBROUTINE create_error
```

- SUBROUTINE **error_constructor**(ifail,info,method): it checks if the optional input `info` and `method` are present and assign them to the `ifail` variable. Then, the SUBROUTINE **error_write** is called. At the end, the execution of the program is stopped if an error is encountered;

```
SUBROUTINE error_constructor(ifail,info,method)
    type(error), intent(out) :: ifail
    character(len=*), intent(in), optional :: info
    character(len=*), intent(in), optional :: method

    if( present(info) ) then
        if( len_trim(info) > 0 ) then
            ifail%info = info
        end if
    end if

    if( present(method) ) then
        if( len_trim(method) > 0 ) then
            ifail%method = method
        end if
    end if

    call error_write(ifail)
    print *, ''
    print *, achar(27)//"[1;91m"// "INTERRPUTING PROGRAM!"//achar(27)//"[0m"
    STOP
END SUBROUTINE error_constructor
```

- SUBROUTINE **error_write**(ifail): it takes as input the `ifail` variable and prints strings `info` and `method` if present. In particular, strings are printed with the addiion of HTML code for using colors in the comand prompt.

```
SUBROUTINE error_write(ifail)
    type(error) :: ifail

    if( len_trim(ifail%info) > 0 ) then
        print *, achar(27)//"[1;91m"// "ERROR: "//achar(27)//"[0m", achar(27)//"[36m"//
    ifail%info //achar(27)//"[0m"
    else
        print *, achar(27)//"[1;91m"// "ERROR: "//achar(27)//"[0m", 'not specified.'
    end if

    if( len_trim(ifail%method) > 0 ) then
        print *, achar(27)//"[1;32m"// "METHOD: "//achar(27)//"[0m", achar(27)//"[36m"//
    ifail%method //achar(27)//"[0m"
    else
        print *, achar(27)//"[1;32m"// "METHOD: "//achar(27)//"[0m", 'not specified.'
    end if

END SUBROUTINE error_write
```

On the other hand, the module `MODULE` **`ERROR_HANDLING_COMMON_ERRORS`** contains:

- `SUBROUTINE` **`error_neg_var`**`(var,dim,ifail,info,method)`: it takes as input a variable `var` of dimension `dim` and checks if the elements are negative. If at least one negative element is found, it returns an error `ifail` with its `info` and `method`;

```
SUBROUTINE error_neg_var(var,dim,ifail,info,method)
    character(len=*), intent(in), optional :: info
    character(len=*), intent(in), optional :: method
    type(error), intent(out), optional :: ifail
    integer :: dim
    integer, dimension(dim), intent(in) :: var
    integer :: ii

    DO ii=1,dim
        IF (var(ii)<0) THEN
            call create_error(ifail,info,method)
        END IF
    END DO

END SUBROUTINE error_neg_var
```

- `SUBROUTINE` **`error_nequal_dim`**`(N1,N2,ifail,info,method)`: it takes as input two variables `N1` and `N2` and check if they are not equal. In this case, an error `ifail` is returned with its `info` and `method`;

```
SUBROUTINE error_nequal_dim(N1,N2,ifail,info,method)
    character(len=*), intent(in), optional :: info
    character(len=*), intent(in), optional :: method
    type(error), intent(out), optional :: ifail
    integer :: N1, N2

    IF (N1.NE.N2) THEN
            call create_error(ifail,info,method)
    END IF
END SUBROUTINE error_nequal_dim
```

- `SUBROUTINE` **`error_kind`**`(ELEM,ELEM_KIND,ifail,info,method)`: it takes as input two elements kind `ELEM` and `ELEM_KIND` and check if they are not equal. In this case, an error `ifail` is returned with its `info` and `method`;

```
SUBROUTINE error_kind(ELEM,ELEM_KIND,ifail,info,method)
    character(len=*), intent(in), optional :: info
    character(len=*), intent(in), optional :: method
    type(error), intent(out), optional :: ifail
    integer, intent(in) :: ELEM_KIND
    integer, intent(in) :: ELEM

    IF(ELEM.NE.ELEM_KIND) THEN
            call create_error(ifail,info,method)
    END IF
END SUBROUTINE error_kind
```

B. Updating Matrix Type Module

We update the MODULE **MATRIX_TYPE** (developed in the last report) with the use of error handling framework previously defined. In particular, we add some checks in the already defined subroutine as the one which initializes or delete the matrix, or the one which computes the trace. If the checks are not satisfied, an error is returned. For instance, in the SUBROUTINE **MatInit**, the dimensions of the matrix N given as an input are checked:

```
call error_neg_var(N,2,ifail,'Matrix dimension must be > 0','MatInit')
```

If one of the elements of N is less than zero, an error occurs. Or, in the SUBROUTINE **MatTrace**, if the trace of a non square matrix is computed, an error occurs:

```
call error_nequal_dim(MServ%N(1),MServ%N(2),ifail,'Trace of a not square matrix','MatTrace')
```

Moreover, we define a new SUBROUTINE **MatMult** for computing the matrix-matrix multiplication by rows with some checks:

- we check if the dimension of the matrices is major than zero, otherwise an error occurs;
- we check if the matrix product for the given dimension is allowed;
- we multiply the two matrices A and B, returning C;
- we check if the dimensions of C are correct;
- we check if C element kind is the same of A (or B).

```
SUBROUTINE MatMult(MServA,MServB,MServC)
    ...
    call error_neg_var(...)
    call error_nequal_dim(...)
    ...multiplication...
    call error_nequal_dim(...)
    call error_kind(...)
    RETURN
END SUBROUTINE MatMult
```

C. Debugging Module

Then, we develop a simple MODULE **DEBUG_CHECKPOINT** to debug the code. It uses the previously defined MODULE **ERROR_HANDLING**, MODULE **ERROR_HANDLING_COMMON_ERRORS** and in addition the just updated MODULE **MATRIX_TYPE**. In particular, this module contains:

- SUBROUTINE **do_checkpoint**(debug,cp_name): it takes as input a logical variable debug and a string cp_name which contains the name of the checkpoint. An integer cp_number, which take count of the number of checkpoint from the starting of the program, is initialized to the value of 1. Also a time variable is initialized to zero. Then, if the logical variable debug is true, the number and current time associated to the checkpoint are printed in the terminal with its associated name;

```
SUBROUTINE do_checkpoint(debug,cp_name)
  implicit none
  logical, intent(in) :: debug
    character(len=*), intent(in), optional :: cp_name
    integer(4) :: cp_number=1
    real(8) :: time=0
    IF (debug) THEN
        IF(present(cp_name)) THEN
            print *, achar(27)//"[1;34m"// "CHECKPOINT: "//achar(27)//"[0m", cp_name
        ELSE
            print *, achar(27)//"[1;34m"// "CHECKPOINT: "//achar(27)//"[0m"
        END IF
        call cpu_time(time)
        print *, "Number: ", cp_number
        print *, "Time  : ", time
        cp_number = cp_number + 1
        print *, ''
    END IF
END SUBROUTINE do_checkpoint
```

- SUBROUTINE **check_matrix**(debug,MServ,cp_name): it takes as input a variable MServ of matrix type. In particular, the dimensions of this matrix and its kind are checked and printed.

```
SUBROUTINE check_matrix(debug,MServ,cp_name)
    implicit none
    logical, intent(in) :: debug
    TYPE(Matrix), INTENT(IN):: MServ
    character(len=*), intent(in), optional :: cp_name
    type(error) :: ifail
    IF (debug) THEN
        call do_checkpoint(debug,cp_name)
        call error_neg_var(MServ%N,2,ifail,'matrix dimension must be > 0')
        print *, "Matrix Dimension: ", MServ%N(1), 'x', MServ%N(2)

        call error_kind(kind(MServ%Elem),ELEM_KIND,ifail,'different type.')
        print *, "Element kind    : ", kind(MServ%Elem)

        IF ( MServ%N(1)==MServ%N(2) ) THEN
            print *, "Trace           : ", MServ%Tr
        ELSE
            print *, "Trace           : ", 'not square matrix.'
        END IF
        print *, ''
    END IF
END SUBROUTINE check_matrix
```

D. Demo Program

We test the developed module with a demo program in which a matrix-matrix multiplication ($A \times B = C$) is performed by using the user-defined SUBROUTINE **MatMult**. In particular, the dimensions of matrices A and B are given as input and they are randomly initialized, while matrix C is initialized with zeros. Then, if the debug option is true, we check matrices A and B by means of SUBROUTINE **check_matrix**: hence, their dimensions and kind are printed. After that, matrix-matrix multiplication is performed and the resulting matrix C (again, if the debug option is true) is also checked. Finally, all the three matrices are deleted.

E. Compilation Stage

The entire code can be easily compiled with the use of a makefile. As said, at the compilation stage the desired **ELEM_TYPE** can be chosen by selecting the option **D** for double real and **Z** for double complex. Moreover, the **-debug** flag has to be added in the demo program to enable debugging.

III. RESULTS

We run the demo program for different sizes of the matrix. In particular, we note that if, by mistake, we insert a negative matrix dimension, an error is returned and the execution of the program is stopped, as wanted. Moreover, the check matrix checkpoint works correctly and print in the terminal useful information about the matrix we are dealing. We also slightly modify the demo program in order to check if all the other implemented and updated subroutines work correctly. We can conlude that the added modules for error handling and debugging behave as wanted.

IV. SELF-EVALUATION

In a further development of the code, it would be useful updating the error handling module by adding error discarding: if a certain error is not problematic, an explicit discarding is needed to ensure that the error is not forgotten somehow. Moreover, it would be useful to print a detailed report of errors in a text file.

REFERENCES

[1] B. V. Koen Poppe, Ronald Cools, (2003).