Assignment 2

Quantum Information & Computing

Francesco Barone

University of Padua, Department of Physics

October 30, 2022

EXR 1 // Checkpoint & debugging

In this exercise, I provide a **FORTRAN module for debugging**.

```
call checkpoint(msg = 'print this message')
```

The **checkpoint** subroutine will print a custom message `msg`. The checkpoints are printed only if a global logical variable `DEBUG` is `true`.

Output

```
[checkpoint] print this message
```

It is possible to **print a variable** alongside the message.

```
real :: x = 1d0
call checkpoint(msg = 'print this message', var = x)
```

Output

```
[checkpoint] print this message
 (real) 1.00000000
```

The optional argument `pause` (logical) will make the checkpoint **wait for user input** to continue.

EXR 1 // Checkpoint & debugging

Additionally, the user can associate each checkpoint to a **verbosity level**.
The global variable `DEBUG_LEVEL` (integer) sets the maximum level of checkpoints that will be printed.

```
call checkpoint(level = 4, msg = 'print this message')
```

For example, this checkpoint will print a message only if the global variable `DEBUG_LEVEL` is ≥ 4.
This feature is useful when you wish to tune how much detailed the debugging checkpoints should be.

I provide the following
standard levels of debugging:

name	level	message printed
DL_CHECK	0	[checkpoint] ...
DL_FATAL	1	[fatal] ...
DL_ERROR	2	[error] ...
DL_WARN	3	[warning] ...
DL_INFO	4	[info] ...
DL_INFOP	5	[info] ...
DL_PEDANTIC	6	[dbg] ...

Example

```
DEBUG = .true.
DEBUG_LEVEL = DL_FATAL
! btw, this is the default value

call checkpoint(level = DL_WARN,
                msg = 'not printed')

DEBUG_LEVEL = DL_PEDANTIC

call checkpoint(level = DL_WARN, msg = 'printed')
```

EXR 2 // Matrix loop with documentation

Documentation

```
!  DOC ============================================================
!
!    This module implements matrix products with nested loops.
!  ------------------------------------------------------------------
!  TYPES:  none
!
!  FUNCTIONS: none
!
!  SUBROUTINES:
!    -  [family] matmul_loop_*    [all of them with same I/O conditions]
!
!                Input   |   m1, m2      input matrices to be multiplied
!                        |   m3          output matrix with the result of m1*m2
!
!           Requirements |   m1, m2, m3 must be allocated before call
!
!                Output  |   none        (implicit result returns to arg m3)
!
!         Post-conditions |   m3 will be overwritten
!
!           Err handlers  |   I will check the correct matrix shapes and use the
!                         |   debugger to notice this error. The product is not
!                         |   executed in this case.
...
```

I've rewritten the module `matmul_loops.f90` with the requested documentation. Checkpoints (from EXR 1) have been included too.

EXR 3 // Complex matrix type

complex8 matrix definition

```
type complex8_matrix
    ! store the dimension of the matrix
    integer, dimension(2) :: size
    ! to store the values of matrix
    complex*8, dimension(:,:), allocatable :: val
end type
```

In the module `mod_matrix_c8.f90` I define the `complex8_matrix` data type, which includes a couple of integer values `size` to keep track of the matrix size, and an allocatable object `val` with 2 dimensions, which will store `complex*8` values.

The matrices can be initialized to zero (i.e. $0 + i0$) or to random complex values:

- call explicitly the subroutine, ex: `CMatInitZero`
- use the interfaces, ex: `.randInit.`

initialization

```
type(complex8_matrix) :: A, B
A = CMatInitZero(shape=(/100, 150/) )
B = .randInit.(/100, 100/)  ! square matrix
```

Math operations

```
complex*8 :: x
x = .Tr.B
A = .Adj.B
```

It is possible to compute the **Trace** of the matrix (only if matrix is square, otherwise an error is risen) and the **Adjoint**.

EXR 3 // Complex matrix type

I provide a routine to **write** the matrix to txt files, as well as a function to **read** matrices from file.

dump and read from file

```
! dumping a matrix to file
call CMatDumpTXT(A, 'data/matrix.txt')

! loading the same matrix from file
A = CMatLoadTXT('data/matrix.txt')
```

Destructor

```
call CMatDelete(A)
call CMatDelete(B)
```

Eventually, you can free a matrix invoking the method `CMatDelete()`.

I also embedded my **debug module** (from EXR 1) inside the `mod_matrix_c8.f90` module, to keep track, if you wish, of higher levels of debugging checkpoints (ex: memory allocations, file parsing errors, ...).

Check `exr3.f90` for an example of higher verbosity stdout.