ASSIGNMENT 2

Physics of Data – Quantum Information and Computing

A.Y. 2024/2025

Lorenzo Calandra Buonaura

04/11/2024

THEORY

- **Debugger and checkpoints**
  - Debugging is the process of identifying, isolating, and fixing bugs in a program; thus, the debugger is an essential tool that helps in analyzing the code line by line, set breakpoints, inspect variable values, and track the execution flow. Usually, the debugger works only when the “debug-mode” is on, which is used when looking for a specific bug or problem.
  - When using a debugger, one can pause program execution at critical points to verify if the code behaves as expected (for example checking the norm of a normalized vector or that an input value is valid); it is useful to include these checkpoints when programming, as they allow to systematically narrow down the location of errors, without causing unexpected problems that could damage the computer itself (like segmentation faults).

- **Fortran modules**
  - In Fortran, modules are a way to group related procedures, data, and variables for better organization and code reuse: they are used to simplify code structure and make it easier to manage large programs. They are usually used to gather related subroutines/functions or for user defined data types: in this way different types can be used in different programs without needing redefinition.
  - The ‘use <modulename>’ statement allows for importing modules into different program units.
  - When programming modules, documentation is critical, as it helps understand the purpose and function of code, making readability and future maintenance easier.

CODE DEVELOPMENT (1)

- For the debugger, a subroutine `checkpoint()` with different level of verbosity has been developed: an higher verbosity value means a longer output when the checkpoint is reached (even though in this case is only dummy elements, which should be modified depending on the specific use of the debugger). A defualt output is present when the verbosity is not a correct value.

```
module debugger
  implicit none
contains
  subroutine checkpoint(debug, verb, msg, var1, var2, var3)
    logical, intent(in) :: debug
    integer, intent(in) :: verb
    character(len=*), optional :: msg
    real(4), intent(in), optional :: var1, var2, var3

    if (debug) then
      select case (verb)
        case (1)
          if (present(msg)) then
            print *, 'Checkpoint: ', msg
          else
            print *, "Checkpoint reached."
          end if
        case …
         …

end module debugger
```

- The `checkpoint()` function has been then used first in a test program, to check the correct behaviour, then it has been called many times in the other modules written.

- We must notice that the `stop` function has been called after the checkpoint, and not inside of it; the reason is because in this way we can use the same function checkpoint to handle both errors and warnings.

CODE DEVELOPMENT (2)

```
!  SUBROUTINES:
!
!  rowbycolumn(A, B, C)
!
!          Inputs  | A(:,:) (real): first matrix (m x n)
!                  | B(:,:) (real): second matrix (n x l)
!                  | C(:,:) (real): result matrix (m x l). If not
!                  |                initialized to all zeros, a warning
!                  |                occurs, then it's set to 0.0 to
!                  |                proceed.
!                  |
!                  | Correct dimensions of the input matrices are
!                  | checked before the multiplication; in case of
!                  | mismatch, an error occurs.
!
!
!  columnbyrow(A, B, C)
!
!          Inputs  | A(:,:) (real): first matrix (m x n)
!                  | B(:,:) (real): second matrix (n x l)
!                  | C(:,:) (real): result matrix (m x l). If not
!                  |                initialized to all zeros, a warning
!                  |                occurs, then it's set to 0.0 to
!                  |                proceed.
!                  |
!                  | Correct dimensions of the input matrices are
!                  | checked before the multiplication; in case of
!                  | mismatch, an error occurs.
```

- For the matrix multiplication timing module, three subroutines have been implemented: two of them performs the two different multiplication methods (row by column and column by row).
- Some pre-conditions are present, in order to handle possible errors (using the debugger), regarding mis-matching matrix sizes and the correct initialization of the product matrix (all zeros).
- The documentation has been written highlighting the inputs and the code has comments that explain the purpose of each line.

CODE DEVELOPMENT (3)

```
subroutine timing(n)
        …
! Open file for writing (replace mode).
  open(unit=rowbycolumn_file, file='rowbycolumn.txt',
status='replace', action='write', iostat=ios)
  if (ios /= 0) then
    call checkpoint(debug=.TRUE., verb=1, msg="ERROR!!! Unable
to open rowbycolumn log file.")
    stop
  end if
        …
  do fraction = 1, 10 ! Loop 10 times.
    m = max(1, fraction * n / 10)
    allocate(A(m, m), B(m, m), C(m, m)) ! Allocation.
        …
    call cpu_time(start_time)
    call rowbycolumn(A, B, C)
    call cpu_time(end_time)
    elapsed_time = end_time - start_time
    write(rowbycolumn_file, '(I5, F10.6)') m, elapsed_time
        …
    deallocate(A, B, C) ! Deallocation.
        …
  end do
  ! Final print statement.
  print *, "Matrix multiplication tests completed successfully."

end subroutine timing
```

- The last subroutine takes care of the performance study; in particular, for eachused method, the data are saved on a different file (with appropriate error handling if not present).
- In order to study the performance scaling with the matrix dimension *n*, each method is called 10 times, with fractions of *n* up to *n* itself.
- A final print statement checks that no errors occured during the execution.
- Also in this case, a test program was used to check for the expected behaviour.

CODE DEVELOPMENT (4)

- For the `complex_matrix` module, a derived TYPE has been defined (double complex matrix), with fuor attributes: the elements, the size, the trace and the adjoint. The module contains a total of two subroutines, used for initialization and saving to file, and two functions, which are used to compute trace and adjoint.

```
module complex_matrix
         … Documentation …
  use debugger
  implicit none

  ! Definition of the derived type for a double complex matrix.
  type :: ComplexMatrix
    double complex, allocatable :: elements(:, :) ! Matrix elements
    integer, dimension(2) :: size                 ! Matrix size
    double complex :: trace                       ! Matrix trace
    double complex, allocatable :: adjoint(:, :)  ! Matrix adjoint
  end type ComplexMatrix

    ! Interfaces for the trace and adjoint operators.
    interface operator(.Tr.)
      module procedure calculate_trace
    end interface

    interface operator(.Adj.)
      module procedure calculate_adjoint
    end interface

  contains
         …
```

- Two interface operators are defined, which are used in order to access directly and in an easier way the aforementioned functions (for trace and adjoint computation).

- Also in this case, the `debugger` module is imported and used for checkpoints.

- The documentation presents the subroutines and the funcions, highlighting inputs and outputs; a general view of the module is provided too.

CODE DEVELOPMENT (5)

```fortran
function calculate_trace(matrix) result(trace_value)
  type(ComplexMatrix), intent(in) :: matrix
  double complex :: trace_value
  integer :: i

  if (matrix%size(1) /= matrix%size(2)) then
    call checkpoint(debug = .TRUE., verb=1, msg="ERROR!!! Trace can only be computed for square matrices.")
    stop
  end if ! Check for a SQUARE matrix

  trace_value = (0.0, 0.0)
  do i = 1, min(matrix%size(1), matrix%size(2))
      trace_value = trace_value + matrix%elements(i, i)
  end do
end function calculate_trace


function calculate_adjoint(matrix) result(adjoint_mat)
  type(ComplexMatrix), intent(in) :: matrix
  double complex, allocatable :: adjoint_mat(:,:)

  allocate(adjoint_mat(matrix%size(2), matrix%size(1)))
  adjoint_mat = transpose(conjg(matrix%elements))
end function calculate_adjoint
```

- In `calculate_trace()`, first of all we check if the matrix is squared: if not, a checkpoint occurs and the program stops (no trace exists for non square matrices).
- The trace is then computed as the sum of the diagonal elements.

- In `calculate_adjoint()`, a transposted matrix is allocated (inverted dimensions).
- The adjoint is then computed as the transposed conjugate of the matrix.

CODE DEVELOPMENT (6)

```
subroutine initialize_matrix(matrix, input_elements)
  implicit none
  type(ComplexMatrix), intent(out) :: matrix
  double complex, allocatable, intent(in) :: input_elements(:, :)
        …      ! Check dimensions of input_elements
        …      ! Set matrix dimensions and allocate memory
  matrix%elements = input_elements
  matrix%trace = calculate_trace(matrix)
  matrix%adjoint = calculate_adjoint(matrix)
end subroutine initialize_matrix


subroutine write_matrix_to_file(matrix, filename)
        …      ! Variable definition
  open(unit=unit_num, file=filename, status='replace', action='write', iostat=ios)
    if (ios /= 0) then
      call checkpoint(debug=.TRUE., verb=1, msg="ERROR: Unable to open file.")
      stop
    end if
        …
    write(unit_num, *) "Matrix Elements" ! Write the matrix elements
      do i = 1, matrix%size(1)
        do j = 1, matrix%size(2)
          write(unit_num, '(F10.4, " + i*", F10.4)', advance='no’)
                 real(matrix%elements(i, j)), aimag(matrix%elements(i, j))
        …
      close(unit_num)
      print *, "Matrix written to file: ", filename
    end subroutine write_matrix_to_file
```

- Initialization has some pre-conditions checks, in order to see if the inputs are ok. The allocates memory, builds the matrix and computes trace and adjoint.

- The saving to file checks first if the file exists, then save the matrix and the other valueable information in a readable format (taking into accounts the fact that are complex numbers).