Quantum Information and Computing

Academic Year 2024 - 2025

ASSIGNMENT 2

Due by October 28, 2024

Gabriel Amorosetti

gabriel.amorosetti@studenti.unipd.it

1. Implementing a debugging subroutine

- Creation of a `module debugger`, that will be used later in the main programs, that contains a checkpoint subroutine

- This subroutine, when called, will output :

  - Nothing, if the logical variable `debug` is not set to `.true.`

  - If set to `.true.`, according to the level of verbosity `verb` set by the user (set to 2 by default) :

    - A simple message to tell that the subroutine was called during execution

    - A specific message set by the user

    - A variable from the main program chosen by the user

```fortran
subroutine checkpoint(debug, verb, msg, var)
    logical :: debug
    integer, optional :: verb
    character(len=*), optional :: msg
    integer, optional :: var

    if (debug) then

        if (verb >= 0) then
            print *, " "
            print *, "Checkpoint called"
            print *, " "
        end if

        if (verb >= 1) then
            print *, msg
        end if

        if (verb >= 2) then
            print *, var ! var needs to be integer !
        end if

    else
        ! no print

    end if
end subroutine checkpoint
```

2. Code quality practices

- The 3rd exercise of the 1st assignment was rewritten in order to include :

  - Documentation : written explanations added in the code to explain its purpose, structure and functionality
  - Comments : descriptions of specific lines to explain in detail what a specific part of the code does and make it clear to maintain
  - Pre- and post-conditions : check a condition before or after executing the main part of the code :
    - Example of post-condition :

```
subroutine check_post_conditions(Z, m)
    integer, intent(in) :: Z(:,:) ! Output of the multiplication
    integer, intent(in) :: m      ! Input matrices size
    ! Subroutine used to check that the output matrix has good dimensions
    if (size(Z, 1) /= m .or. size(Z, 2) /= m) then
        print *, "Error: Result matrix has incorrect dimensions."
    else
        print *, "Post-condition: Result matrix has correct dimensions."
    end if
end subroutine
```

  - Error handling : detect and manage errors during execution
    - Example of pre-condition with error handling :

```
! Pre-condition to check that n (Matrices size) is superior or equal to 1
if (n < 1) then
    print *, "Error, n must be superior or equal to 1"
    stop    ! Error handling to stop the program if the input is incorrect
end if
```

  - Checkpoints : intermediate validation steps to monitor the execution
    - Implemented using the subroutine of the 1st exercise :

```
C = matmul(A,B)

call checkpoint(.true., 1, "Multiplication done")
```

3. Derived types : double precision complex matrix

- Implementation of a `module mod_matrix_c8` to define :
  - A double complex matrix derived type

```
type :: complex8_matrix
      integer, dimension(2) :: size     ! Size of the matrix (array of dimension 2 representing the number of rows and columns)
      complex*8, dimension(:,:), allocatable :: elem ! Elements of the matrix
end type complex8_matrix
```

  - A subroutine initializing this new type

```
subroutine randInit(cmx, dims)

      type(complex8_matrix), intent(out) :: cmx   ! Complex matrix to initialize
      integer, dimension(2), intent(in) :: dims     ! Size of the matrix (array of dimension 2 representing the number of rows and columns)
      real(8), allocatable :: real_part(:,:), imag_part(:,:) ! Real and imaginary parts of the complex element

      cmx%size = dims      ! Assign the input dimensions to the complex matrix

       allocate(cmx%elem(dims(1), dims(2)))    ! Allocation of the matrix elements
       allocate(real_part(dims(1), dims(2)))    ! Allocation of the real parts
       allocate(imag_part(dims(1), dims(2)))    ! Allocation of the imaginary parts

       call random_seed()                  ! Initialization of the real and imaginary parts with random values (between 0 and 1)
       call random_number(real_part)
       call random_number(imag_part)

       ! Combining these parts into complex matrix elements and scaling values by 100.0
       cmx%elem = cmplx(real_part * 100.0, imag_part * 100.0, kind=8)

       deallocate(real_part, imag_part)    ! Deallocation of the real and imaginary parts
end subroutine randInit
```

3. Derived types : double precision complex matrix

- Implementation of a `module mod_matrix_c8` to define :
  - The functions Trace and Adjoint

```fortran
function CMatTrace(cmx) result(tr)

        type(complex8_matrix), intent(in) :: cmx ! Input matrix
        complex*8 :: tr   ! Output (trace)
        integer :: ii

        tr = complex(0.0, 0.0)  ! Initialization to zero

        ! Summing up diagonal elements
         do ii = 1, cmx%size(1)
            tr = tr + cmx%elem(ii, ii)
         end do

 end function
```

```fortran
function CMatAdjoint(cmx) result(cmxadj)

        type(complex8_matrix), intent(in) :: cmx ! Input matrix
        type(complex8_matrix) :: cmxadj            ! Output matrix

        ! Setting size of the adjoint matrix (transpose of input dimensions)
        cmxadj%size(1) = cmx%size(2)
        cmxadj%size(2) = cmx%size(1)

         ! Allocating adjoint matrix elements :
         allocate(cmxadj%elem(cmxadj%size(1), cmxadj%size(2)))
         ! Computing the conjugate transpose of the elements :
         cmxadj%elem = conjg(transpose(cmx%elem))
 end function
```

  - And their correspondent Interfaces

```fortran
interface operator(.Tr.)

        module procedure CMatTrace

end interface
```

```fortran
interface operator(.Adj.)

        module procedure CMatAdjoint

end interface
```

3. Derived types : double precision complex matrix

- Implementation of a `module mod_matrix_c8` to define a subroutine that writes on a text file the matrix type in a readable form :

```fortran
subroutine CMatDumpTXT(cmx, file_name)

        type(complex8_matrix), intent(in) :: cmx    ! Input matrix
        character(len=*), intent(in) :: file_name    ! Output file name
        integer :: i, j

        open(unit=10, file=file_name, status='replace', action='write')

         ! Writing matrix size
         write(10, *) 'Matrix Dimensions: ', cmx%size(1), 'x', cmx%size(2)

         ! Writing each row of the matrix
         do i = 1, cmx%size(1)
             ! Adjust format to output each element in a row,
             ! with a space between elements
             write(10, '(100F10.4)') (cmx%elem(i, j), j = 1, cmx%size(2))
         end do

         close(10)

 end subroutine CMatDumpTXT
```

- Including everything in a program

```fortran
program main

    use mod_matrix_c8
    implicit none

    type(complex8_matrix) :: A, C  ! Defining the complex matrices
    complex*8 :: x                 ! Trace of the matrix
    integer :: dimensions(2)       ! Matrix size (rows and columns)

     ! Size of the matrix
     dimensions = [100, 100]

     ! Matrix initialization
     call randInit(A, dimensions)

     ! Trace of the matrix (square matrix only)
     x = .Tr. A

     ! Adjoint matrix
     C = .Adj. A

     ! Writing the matrix on a text file :
     call CMatDumpTXT(C, 'matrix_output.txt')
     print *, "The adjoint matrix has been written to 'matrix_output.txt'."

 end program main
```