QUANTUM INFORMATION AND COMPUTING

ASSIGNMENT 2

Physics of Data

Turci Andrea

November 2nd, 2024

1. CHECKPOINTS

Module for checkpoint functionality

- Providing different subroutines for real, integer and character variables, through a core subroutine.
- Includes control via a logical variable (Debug=.TRUE. or .FALSE.)
- Offers verbosity levels (controlled by optional integer 'verbosity' parameter)
  - Level 1: Basic checkpoint message.
  - Level 2: Detailed checkpoint with optional variable printout.
  - Level 3: Full verbosity message with additional variable printout.
- Allows printing of optional, user-defined message and variables.

3 Subroutines handling different data types:

```
subroutine checkpoint_real(debug, verbosity, msg, var1, var2, var3)
    ...
    call checkpoint_core(debug, verbosity, msg, var1, var2, var3)
end subroutine checkpoint_real
```

```
subroutine checkpoint_integer(debug, verbosity, msg, var1, var2, var3)
    ...
    call checkpoint_core(debug, verbosity, msg, var1, var2, var3)
end subroutine checkpoint_integer
```

```
subroutine checkpoint_character(debug, verbosity, msg, var1, var2, var3)
    ...
    call checkpoint_core(debug, verbosity, msg, var1, var2, var3)
end subroutine checkpoint_character
```

Subroutine to print the variable according to the type

```
subroutine print_variable(var, label)
        class(*), intent(in) :: var
        character(len=*), intent(in) :: label

        select type(var)
            type is (real(8))
                print*, label, var
            type is (integer)
                print*, label, var
            type is (character(len=*))
                print*, label, trim(var)
            class default
                print*, label, 'Unknown data type'
        end select
    end subroutine print_variable
```

```fortran
subroutine checkpoint_core(debug, verbosity, msg, var1, var2, var3)
   ...
    if (debug) then
        if (vlevel == 1) then
            if (present(msg)) then
                print*, 'Checkpoint:', trim(msg)
            else
                print*, 'Checkpoint: Debugging checkpoint reached.'
            end if
        end if

        if (vlevel == 2) then
            if (present(msg)) then
                print*, 'Detailed Checkpoint:', trim(msg)
            else
                print*, 'Detailed Checkpoint: Debugging checkpoint reached.'
            end if
            if (present(var1)) call print_variable(var1, 'time = ')
        end if

        if (vlevel == 3) then
            if (present(msg)) then
                print*, 'Full details:', trim(msg)
            else
                print*, 'Fully detailed Checkpoint: Debugging checkpoint reached.'
            end if
            if (present(var1)) call print_variable(var1, 'n_size = :')
            if (present(var2)) call print_variable(var2, 'rows = ')
            if (present(var3)) call print_variable(var3, 'cols = ')
        end if

        if (vlevel > 3) then
            print*, 'Invalid verbosity value. Choose between 1, 2, and 3.'
        end if
    end if
end subroutine checkpoint_core
```

1. CHECKPOINTS

`checkpoint_core` subroutine to handle verbosity and debugging output.

Parameters:

- `debug`: logical (true/false) to control whether debug output is active
- `verbosity`: integer (optional) to handle verbosity by setting the level of detail (1, 2 or 3).
- `msg`: optional message to customize the checkpoint output
- `var1, var2, var3`: optional variables to print based on verbosity

- According to different verbosity level we are able to chose whether printing a user-defined message or a pre-defined one.
- If the variables are given in input we are able to print them, calling the previous functions.
- Details increase by increasing the verbosity value
- Error checking: if the verbosity is greater than 3, a printed message will highlight this

```fortran
do
    print*, "Enter max_size (default 900):"
    read(*, *, IOSTAT=io_status) max_size
    if (io_status == 0 .and. max_size > 0) exit
    print*, "Invalid input. Please enter a positive integer for max_size."
    max_size = 900  ! Default value
end do
do
    print*, "Enter step (default 100, must be less than max_size):"
    read(*, *, IOSTAT=io_status) step
    if (io_status == 0 .and. step > 0 .and. step < max_size) exit
    print*, "Invalid input. Please enter a positive integer less than max_size."
end do
do
    print*, "Enter seed (default 12345):"
    read(*, *, IOSTAT=io_status) seed
    if (io_status == 0 .and. seed > 0) exit
    print*, "Invalid input. Please enter a positive integer for seed."
    seed = 12345  ! Default value
end do
do
    print*, "Enter optimization flag (O1, O2, O3; default O2):"
    read(*, '(A)', IOSTAT=io_status) opt_flag
    opt_flag = trim(adjustl(opt_flag))
    if (io_status == 0 .and. (opt_flag == "O1" .or. opt_flag == "O2" .or. opt_flag == "O3")) exit
    print*, "Invalid input. Please enter one of O1, O2, O3."
    opt_flag = "O2"  ! Default value
end do
do
    print*, "Enter type of multiplication (matmul, row-col, col-row, ALL; default ALL):"
    read(*, '(A)', IOSTAT=io_status) type_mult
    type_mult = trim(adjustl(type_mult))
    if (io_status == 0 .and. (type_mult == "matmul" .or. type_mult == "row-col" &
        .or. type_mult == "col-row" .or. type_mult == "ALL")) exit
    print*, "Invalid input. Please enter one of matmul, row-col, col-row, ALL."
    type_mult = "ALL"  ! Default value
end do
```

2. DOCUMENTATION

Program to compare the performance for

- row-by-column multiplication
- column-by-row multiplication
- MATMUL multiplication

Parameters are asked in input:

1. Maximum matrix size (`max_size`)
2. Step size (`step`):
3. Seed for random number generator (`seed`)
4. Optimization flag (`opt_flag`
5. Type of multiplication (`type_mult`)

For each input:

- parameters conditional statements are made to verify and ensure valid input is provided for each parameter, otherwise errors are thrown
- Default values are properly chosen

2. DOCUMENTATION

```
subroutine perform_multiplications(max_size, step, seed, opt_flag, type_mult)
   ...
    ! Preconditions
    call checkpoint_real(debug=.TRUE., msg='Beginning matrix multiplication process.')
    if (max_size <= 0 .or. step <= 0 .or. step >= max_size) then
        print*, "Error: Invalid matrix size or step configuration."
        return
    end if

    call prepare_output_file(filename, type_mult, max_size, opt_flag, step)
```

```
subroutine prepare_output_file(filename, type_mult, max_size, opt_flag, step)
    ...
    write(filename, '(A, A, A, A, A, A)') "data_mult/" // trim(type_mult) // "_size_", &
        trim(max_size_str), "_" // trim(opt_flag) // "_step_", trim(step_str) // ".dat"

    ! Check if file exists, and if not, create it with the header
        inquire(file=filename, exist=flag)
        if (.not. flag) then
            open(unit=20, file=filename, status="replace", action="write")
            if (type_mult == "ALL") then
                write(20, '(A)') 'Explicit(i-j-k)    Column-major(i-k-j)    MATMUL'
            else if (type_mult == "row-col") then
                write(20, '(A)') 'Explicit(i-j-k)'
            else if (type_mult == "col-row") then
                write(20, '(A)') 'Column-major(i-k-j)'
            else if (type_mult == "matmul") then
                write(20, '(A)') 'MATMUL'
            end if
        end if
end subroutine prepare_output_file
```

`perform_multiplications` performs the matrix multiplications based on the specified parameters and measures the time taken for each method.

Checkpoint: to notify that the matrix multiplication process has started. No details needed.

Pre-condition: to ensure that the parameter max_size is positive, and that the increment size for the matrix dimension is smaller than max_size and positive as well.

`prepare_output_file` prepares the output file:

- The name of the file contains all the necessary parameters to identify the matrix multiplication. For example:

  row-col_size_800_O3_step_200.dat

- Write and prepare the header depending on the multiplication method applied

2. DOCUMENTATION

```
if (type_mult == "ALL" .or. type_mult == "row-col") then  # row-by-column method

    call cpu_time(start_time)
    call matrix_multiply_explicit(A, B, C_explicit, i)
    call cpu_time(end_time)

    time_explicit = end_time - start_time
    call checkpoint_real(debug = .TRUE., verbosity= 2, msg = 'Time taken for row-col method', var1 = time_explicit)

end if
```

Depending on the method (and so the `type_mult` parameter) the cpu time is computed.

- A checkpoint at verbosity 2 notifies and print the cpu time taken to perform the operation.

When calling the multiplication subroutine, pre-conditions and post-conditions are applied before and after the matrix multiplication:

- Pre-condition: A checkpoint prints the current size of the result matrix. Afterwards, I verify that the size of input and output matrices is the expected one
- Post-condition: I call the same checkpoint and the same conditional statement to verify whether or not the size of one of the matrices has changed during the operation, resulting in an error.

```
! Preconditions check
call checkpoint_integer(debug = .TRUE., verbosity = 3, msg = 'Starting row-col multiplication, with ', var1 = size(C,1))

if (size(A,1) /= n .or. size(A,2) /= n .or. size(B,1) /= n .or. size(B,2) /= n .or. size(C,1) /= n .or. size(C,2) /= n) then
  print*, "Error: Invalid matrix dimensions for explicit multiplication."
  return
end if
```

```
! Post-conditions check
call checkpoint_integer(debug = .TRUE., verbosity = 3, msg = 'Finishing row-col multiplication, with ', var1 = size(C,1))

if (size(A,1) /= n .or. size(A,2) /= n .or. size(B,1) /= n .or. size(B,2) /= n .or. size(C,1) /= n .or. size(C,2) /= n) then
   print*, "Error: Invalid matrix dimensions for explicit multiplication."
   return
end if
```

2. DOCUMENTATION

Output:
For each step size for matrix size increments I have an output of this kind. All the information needed through the operation of matrix multiplication are provided. The error handling and the checkpoints defined in the debugger work properly:

- The size before and after the matrix multiplication is the same
- The CPU time at each step is provided

```
------------------------
Matrix size:         150
Full details:Starting row-col multiplication, with
n_size = :         150
Full details:Finishing row-col multiplication, with
n_size = :         150
Detailed Checkpoint:Time taken for row-col method
time =    1.5616000000000001E-002
Full details:Starting col-row multiplication, with
n_size = :         150
Full details:Finished col-row multiplication, with
n_size = :         150
Detailed Checkpoint:Time taken for col-row method
time =    1.7605999999999997E-002
Detailed Checkpoint:Time taken for intrinsic MATMUL
time =    9.6499999999999364E-004
------------------------
```

File generation:
The file created that collects all the CPU times w.r.t. the different matrix multiplication methods is built in this way:

Explicit(i-j-k)	Column-major(i-k-j)	MATMUL
0.005871	0.005604	0.001186
0.042546	0.037168	0.001420
0.105069	0.109763	0.002403
0.263790	0.305009	0.004769
0.446677	0.583112	0.010965
0.768484	0.979403	0.023605
1.259546	1.676771	0.032258
2.151774	3.177201	0.042249
3.266156	4.357992	0.055755

3. DERIVED TYPES

```fortran
module mod_matrix_c8
    use debugger
    implicit none

    type :: complex8_matrix
        integer, dimension(2) :: size       ! Matrix dimensions (rows, columns)
        complex(8), allocatable :: elem(:,:)    ! Matrix elements
    end type complex8_matrix

    interface operator(.Adj.)
        module procedure CMatAdjoint    ! Operator overload for adjoint (.Adj.)
    end interface operator(.Adj.)

    interface operator(.Tr.)
        module procedure CMatTrace      ! Operator overload for trace (.Tr.)
    end interface operator(.Tr.)
contains
```

Module `mod_matrix_c8`:

- Defines the derived type '`complex8_matrix`' to handle double complex matrices
- Provides subroutines and functions for
    - ➢ initialization,
    - ➢ adjoint (conjugate transpose),
    - ➢ trace calculation,
    - ➢ equality checking,
    - ➢ file output.

```fortran
subroutine initMatrix(cmx, rows, cols)
    type(complex8_matrix), intent(out) :: cmx
    integer, intent(in) :: rows, cols

    ! Debugging checkpoint to track matrix initialization dimensions
    call checkpoint_integer(debug = .true., verbosity = 3,
                        msg = "Initializing matrix", var2 = rows, var3 = cols)

    cmx%size(1) = rows ! Set matrix dimensions
    cmx%size(2) = cols

    ! Allocate memory for the elements array with specified dimensions
    allocate(cmx%elem(rows, cols))

    cmx%elem = (0.0d0, 0.0d0) ! Initialize all elements to (0.0, 0.0)
end subroutine initMatrix
```

Subroutine `initMatrix` :

- Initializes a complex8_matrix instance to specified dimensions
- Input parameters:
    - ➢ `cmx` : Output complex8_matrix to initialize
    - ➢ `rows, cols`: Number of rows and columns for the matrix

- `checkpoint_integer`: Debugging checkpoint to track matrix initialization dimensions
- Initializes all elements to 0

3. DERIVED TYPES

```fortran
function CMatAdjoint(cmx) result(cmxadj)
    type(complex8_matrix), intent(in) :: cmx
    type(complex8_matrix) :: cmxadj

    cmxadj%size(1) = cmx%size(2)
    cmxadj%size(2) = cmx%size(1)

    allocate(cmxadj%elem(cmxadj%size(1), cmxadj%size(2)))

    cmxadj%elem = conjg(transpose(cmx%elem))
end function CMatAdjoint
```

Subroutine `CMatAdjoint`: Computes the adjoint of a complex8_matrix.

- Set the size of the adjoint matrix by transposing the dimensions
- Allocate memory for the adjoint matrix with transposed dimensions
- Compute the adjoint by taking the conjugate transpose of the elements, using `conjg(transpose(cmx%elem))`

```fortran
    function CMatTrace(cmx) result(tr)
        type(complex8_matrix), intent(in) :: cmx
        complex(8) :: tr
        integer :: ii

        tr = (0.0d0, 0.0d0)

        do ii = 1, cmx%size(1)
            tr = tr + cmx%elem(ii, ii)
        end do
    end function CMatTrace
```

Subroutine `CMatTrace`: Calculates the trace of a square complex8_matrix

- Initialize trace to zero
- Sum the diagonal elements to compute the trace

3. DERIVED TYPES

```fortran
subroutine CMatDumpTXT(cmx, cmx_adjoint, trace_cmx, trace_cmx_adjoint, seed, filename)
   ...
    write(filename, '(A, A, A, A, A, A)') "complex_matrix/matrix_result_" // trim(dim_1) // "x", &
        trim(dim_2), "_seed_" // trim(char_seed) // ".dat"

    open(unit=10, file=filename, status='replace', iostat=i)
    if (i /= 0) then
        print *, "Error opening file: ", filename
        return
    end if

    write(10, *) "Matrices Size: ", cmx%size(1), " x ", cmx%size(2)
    write(10, *) "ORIGINAL MATRIX:"
    write(10, *) "Trace: ", trace_cmx
    write(10, *) "Elements:"

    do i = 1, cmx%size(1)
        do j = 1, cmx%size(2)
            write(10, '(A)', advance='no') '('
            write(10, '(F7.4, ",", F7.4)', advance='no') real(cmx%elem(i, j)), aimag(cmx%elem(i, j))
            write(10, '(A)', advance='no') ')'
            if (j < cmx%size(2)) write(10, '(A)', advance='no') ' '  ! Separate elements with space
        end do
        write(10, *)  ! New line after each row
    end do
    close(10)

    ! The same procedure is performed for the Adjoint Matrix

    call checkpoint_character(debug = .true., verbosity = 1, msg = "Matrix written to file", var1 = filename)
end subroutine CMatDumpTXT
```

Subroutine `CMatDumpTXT`: write matrix on file

- Write the file name. It can be for example:

  matrix_result_4x4_seed_55.dat

- Open the file for writing and checks if some errors have occurred
- Write matrix dimensions and trace to file
- Write elements of the original matrix
- Same procedure of writing into the file is done for the Adjoint matrix
- Debugging checkpoint to confirm file writing

3. DERIVED TYPES

Program main:

1. Asks for the parameters in input
2. Initializes the matrix, generating random real and imaginary parts.
   Computes the trace and checks if it is correct
3. Verifies the following relation $(A^H)^H = A$
4. Verifies the following relation $tr(A^H) = \overline{tr(A)}$

1.

```
do
    print*, "Enter size of the matrix (default 3):"
    read(*, *, IOSTAT=io_status) size
    if (io_status == 0 .and. size > 0) exit
    print*, "Invalid input. Please enter a positive integer for size."
    size = 3  ! Default value
end do
```

2.

```
call initMatrix(A, rows, cols)
 ...
call random_number(A%elem(i, j)%re)
call random_number(A%elem(i, j)%im)
A%elem(i, j) = dcmplx(A%elem(i, j)%re, A%elem(i, j)%im)
 ...
trace_A = .Tr. A
 ...
checkpoint_character(debug = .true., verbosity = 1,
             msg = "Trace of A is incorrect")
```

3.

```
A_adjoint = .Adj. A
trace_A_adjoint = .Tr. A_adjoint

A_adjoint_adjoint = .Adj. A_adjoint
...
call matrices_are_equal(A, A_adjoint_adjoint, areEqual)
...
call checkpoint_character(debug = .true., verbosity = 1,
             msg = "Verification: (A^H)^H = A is incorrect")
```

4.

```
trace_A_conjugate = .Tr. A_adjoint

call checkpoint_character(debug = .true., verbosity = 1,
             msg = "Verification: tr(A^H) = conjugate of tr(A) is incorrect")
```

Output:

```
Full details:Initializing matrix
rows =            5
cols =            5
Trace of matrix A:             (3.6928475373314198,2.1190493082894108)
Checkpoint:Trace of A is correct
Checkpoint:Verification: (A^H)^H = A is correct
Checkpoint:Verification: tr(A^H) = conjugate of tr(A) is correct
Checkpoint:Matrix written to file
The original matrix has been written to file: complex_matrix/matrix_result_5x5_seed_333.dat
```

Example of file created:

```
 Matrices Size:           5  x           5
 ORIGINAL MATRIX:
 Trace:               (2.6987591073199746,1.8994984603549057)
 Elements:
( 0.4710, 0.1172) ( 0.7247, 0.2761) ( 0.7073, 0.3703) ( 0.8962, 0.7217) ( 0.4310, 0.6008)
( 0.7166, 0.9682) ( 0.5189, 0.9373) ( 0.2051, 0.0299) ( 0.7369, 0.0272) ( 0.4202, 0.0648)
( 0.9136, 0.2459) ( 0.2572, 0.2643) ( 0.7710, 0.1192) ( 0.1828, 0.9410) ( 0.0096, 0.0492)
( 0.7805, 0.5630) ( 0.0729, 0.4286) ( 0.7803, 0.0521) ( 0.2017, 0.6964) ( 0.9246, 0.2011)
( 0.3978, 0.1761) ( 0.6331, 0.6753) ( 0.5798, 0.3451) ( 0.6040, 0.9854) ( 0.7362, 0.0294)

 ADJOINT MATRIX:
 Trace:              (2.6987591073199746,-1.8994984603549057)
 Elements:
( 0.4710,-0.1172) ( 0.7166,-0.9682) ( 0.9136,-0.2459) ( 0.7805,-0.5630) ( 0.3978,-0.1761)
( 0.7247,-0.2761) ( 0.5189,-0.9373) ( 0.2572,-0.2643) ( 0.0729,-0.4286) ( 0.6331,-0.6753)
( 0.7073,-0.3703) ( 0.2051,-0.0299) ( 0.7710,-0.1192) ( 0.7803,-0.0521) ( 0.5798,-0.3451)
( 0.8962,-0.7217) ( 0.7369,-0.0272) ( 0.1828,-0.9410) ( 0.2017,-0.6964) ( 0.6040,-0.9854)
( 0.4310,-0.6008) ( 0.4202,-0.0648) ( 0.0096,-0.0492) ( 0.9246,-0.2011) ( 0.7362,-0.0294)
```