Debugging

Abstract

The aim of this exercise is to implement the first week exercise on matrix multiplication including a subroutine for debugging, comments, pre/post conditions, error handling, checkpoints and a documentation.

1 Theory

Matrix multiplication is a binary operation that produces a matrix from two matrices. If A is an $m \times n$ matrix and B is an $n \times p$ matrix,

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathrm{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the matrix product C = AB is defined to be the $m \times p$ matrix

$$\mathrm{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$. So, matrix multiplication is well defined only if the number of columns in the first matrix is equal to the number of rows in the second matrix.

2 Code development

In order to manage debugging we define a dedicated module, called `debugger`.

```
module debugger
    ! module for debugging
    ! It contains a subroutine that can be used as checkpoint for debugging.
    implicit none
contains
    subroutine checkpoint(debug, variable, message, end_program)
        implicit none
        ! debug: logical variable that enables debugging if it is .true.
        ! end_program: logical variable that stops the program if is .true.
        logical, intent(in) :: debug, end_program
        ! optional variable to be printed
        class(*), intent(in), optional :: variable
        ! optional array to be printed
        class(*), dimension(:), intent(in), optional :: array_variable
        ! optional message to be printed
        character(*), optional :: message
```

```
        ! if debug = true then the message will be printed
        ! and in case of presence of a variable it will also print it
        if (debug) then
            print *, "[Debugger] --------------------------"
            print *, message
            if (present(variable)) then
                select type(variable)
                    type is (integer(2))
                        print *, variable
                    type is (integer(4))
                        print *, variable
                    type is (real(4))
                        print *, variable
                    type is (real(8))
                        print *, variable
                    type is (complex(8))
                        print *, variable
                    type is (complex(16))
                        print *, variable
                    type is (logical)
                        print *, variable
                end select
            end if
            if (present(array_variable)) then
                select type(array_variable)
                    type is (integer(2))
                        print *, array_variable
                    type is (integer(4))
                        print *, array_variable
                    type is (real(4))
                        print *, array_variable
                    type is (real(8))
                        print *, array_variable
                    type is (complex(8))
                        print *, array_variable
                    type is (complex(16))
                        print *, array_variable
                end select
            end if
            print *, "-----------------------------------"
            ! if end_program = true then in case of error the program will stop
            if (end_program) then
                stop
            end if
        end if
    end subroutine checkpoint
end module debugger
```

Inside this module we define a subroutine `checkpoint`, which takes in input the following arguments:

- `debug`, a logical variable that enables debugging;
- `variable`, a (optional) variable to be printed. The printing is done through a `select type` construct, which allows to consider different types of variables. In this specific program we only have `logical`, `integer`, `real` or `character` variables but for sake of generality we also include `complex` variables;
- `array_variable`, a (optional) array to be printed;
- `message`, a (optional) string to be printed;

- `end_program`, a logical variable that can allow the interruption of the program.

If `debug = true` then a message will be printed. If the optional argument `variable` is specified, then it will be printed as well. In case of serious errors, the programmer can set the argument `end_program = true` to stop the program if the error occurs.

In the main program `MyMatrixMultiplication` we use the subroutine `checkpoint` to print a message if the number of columns in the first matrix is different from the number of rows in the second one or if one of the dimensions entered by the user is less than one. Notice that in both these cases `end_program = false`, in fact instead of stopping the program it is better to ask the user to enter again the dimension of the matrices.

```
    ! enter the dimension of the matrix
    print *, "Please insert the dimension of the two matrices to multiply. &
             &Please recall that the number of columns in the first matrix &
             &must be equal to the number of rows in the second matrix."
    print *, "Please enter the dimension of the first matrix [nrows, ncols]:"
    read (*, *) nrows1, ncols1
    print *, "Please enter the dimension of the second matrix [nrows, ncols]:"
    read (*, *) nrows2, ncols2
    ! ask to enter the dimension until nrows and ncols are greater or equal
    !than 1 and nrows2=ncols1
    do while ((nrows2 .ne. ncols1) .or. (((nrows1 .lt. 1) .or. (ncols1 .lt. 1)) .
    or. ((nrows2 .lt. 1) .or. (ncols2 .lt. 1))))
        ! display two different messages according to
        ! the condition which is not satisfied
        ! if nrows2=ncols1 warn the user that the number of
        ! rows of the second matrix must
        ! be equal to the number of columns of the first matrix
        if (nrows2 .ne. ncols1) then
            call checkpoint(debug = .true.,
                            message = "The number of columns in &
                                      &the first matrix is not equal &
                                      &to the number of rows in the &
                                      &second matrix.",
                            end_program = .false.)
            print *, "Please enter the dimension of the first matrix &
                     &[nrows, ncols]:"
            read (*, *) nrows1, ncols1
            print *, "Please enter the dimension of the second matrix &
                     &[nrows, ncols]:"
            read (*, *) nrows2, ncols2
        end if
        ! if some of the dimensions is less than 1 warn the user
        if (((nrows1 .lt. 1) .or. (ncols1 .lt. 1)) .or. ((nrows2 .lt. 1) .or. (
    ncols2 .lt. 1))) then
            call checkpoint(debug = .true.,
                            message = "Non valid dimension! The number of rows &
                                      &and columns must be greater than 1.",
                            end_program = .false.)
            print *, "Please enter the dimension of the first matrix &
                     &[nrows, ncols]:"
            read (*, *) nrows1, ncols1
            print *, "Please enter the dimension of the second matrix&
                     &[nrows, ncols]:"
            read (*, *) nrows2, ncols2
        end if
    end do
```

This is only one particular way of using the `checkpoint` subroutine, that is calling it to display a message when an error raises. In other cases, it can be simply used to check some variables while writing the code, for example printing the name and the value. A trivial example follows.

```
    ...
    print *, "Please enter the dimension of the first matrix [nrows, ncols]:"
    read (*, *) nrows1, ncols1
    call checkpoint(debug = .true., message = "nrows1 =", variable = nrows1)
    call checkpoint(debug = .true., message = "ncols1 =", variable = ncols1)
    ...
```

Another possible use of the `checkpoint` subroutine is to check if the dimension of the matrix product is correct and print its shape. An example follows.

```
    ...
    m3 = mult1(m1, m2)
    ! check if the dimension of m3 is correct
    call checkpoint(debug = (all(shape(m3) == (/nrows1,ncols2/))), &
                    array_variable=shape(m3), &
                    message="m3 has the right shape.", &
                    end_program=.false.)
    ...
```

3 Results

The folder with the code and the executable file contains also a file `Documentation.txt` which contains a reference guide explaining the purpose of the program and how to use it.

When the program is executed, the following line will be printed.

```
    Please enter the dimension of the two matrices to multiply. Please recall that
     the number of columns in the first matrix must be equal to the number of rows
     in the second matrix.
    Please enter the dimension of the first matrix [nrows, ncols]:
```

Once the user entered the dimension of the first matrix, he will be asked the same for the second one.

```
    Please enter the dimension of the second matrix [nrows, ncols]:
```

Depending on the inputs given by the user, there are three possibilities:

1. the number of columns in the first matrix is equal to the number of rows in the second matrix and all the dimensions are greater or equal to one. In this case the program will proceed to the computation of the multiplication (the user will be asked if he/she want to save the results in a text file, warning that in case of large matrices its size can be large);

2. the dimensions are all greater or equal than one but the number of columns in the first matrix is not equal to the number of rows in the second matrix. In this case the user will be notified by the following message:

   ```
    [Debugger] -----------------------------------------------------------
    The number of columns in the first matrix is not equal to the number of
    rows in the second matrix.
    -----------------------------------------------------------------------
   ```

   and will be asked to enter again the dimensions of the two matrices;

3. not all the dimensions are greater or equal than one. In this case the user will be notified by the following message:

   ```
    [Debugger] -----------------------------------------------------------
    Non valid dimension! The number of rows and columns must be greater or
    equal than 1.
    -----------------------------------------------------------------------
   ```

and will be asked to enter again the dimensions of the two matrices.

4 Self evaluation

This exercise was very instructive since it made me understand that debugging is a good programming practice that must always be used, since it improves the reliability of the code.

A possible improvement of this debug module could be adding an internal division for zero inside the subroutine `checkpoint`, in particular inside the `if(end_program)` statement.

```
    subroutine checkpoint(debug, variable, message, end_program)
        ...
        integer :: sp
        ...
        if (debug) then
            ...
            if (end_program) then
                ! intentional division by zero to stop the program
                sp = 1
                sp = 1/(sp-sp)
            end if
        end if
    end subroutine checkpoint
```

If compiled with `gfortran -ffpe-trap=zero` and `-fbacktrace` options, this will stop the program because of the intentional division by zero and a call stack will be printed. An example follows.

```
    Program received signal SIGFPE: Floating-point exception - erroneous
    arithmetic operation.

    Backtrace for this error:
    #0  0x7f85c064bd01 in ???
    #1  0x7f85c064aed5 in ???
    #2  0x7f85c047f20f in ???
    #3  0x55768b12e22d in debugger_MOD_checkpoint
      at /home/prova.f90:20
    #4  0x55768b12e260 in MAIN
      at /home/prova.f90:40
    #5  0x55768b12e2a3 in main
      at /home/prova.f90:31
    Floating point exception (core dumped)
```

Since it is written in hexadecimal, it is not entirely clear how to interpret the backtrace. However, at least we can read the lines corresponding to the error.

Another possible improvement would be to also include a (optional) `matrix_variable` to print a matrix.