Debugging, documenting, and creating types in Fortran

Alberto Salvador
Physics of Data, University of Padua

Course: Quantum information and computing

Assignment 2

Outline of the presentation

- Debugging modules in Fortran

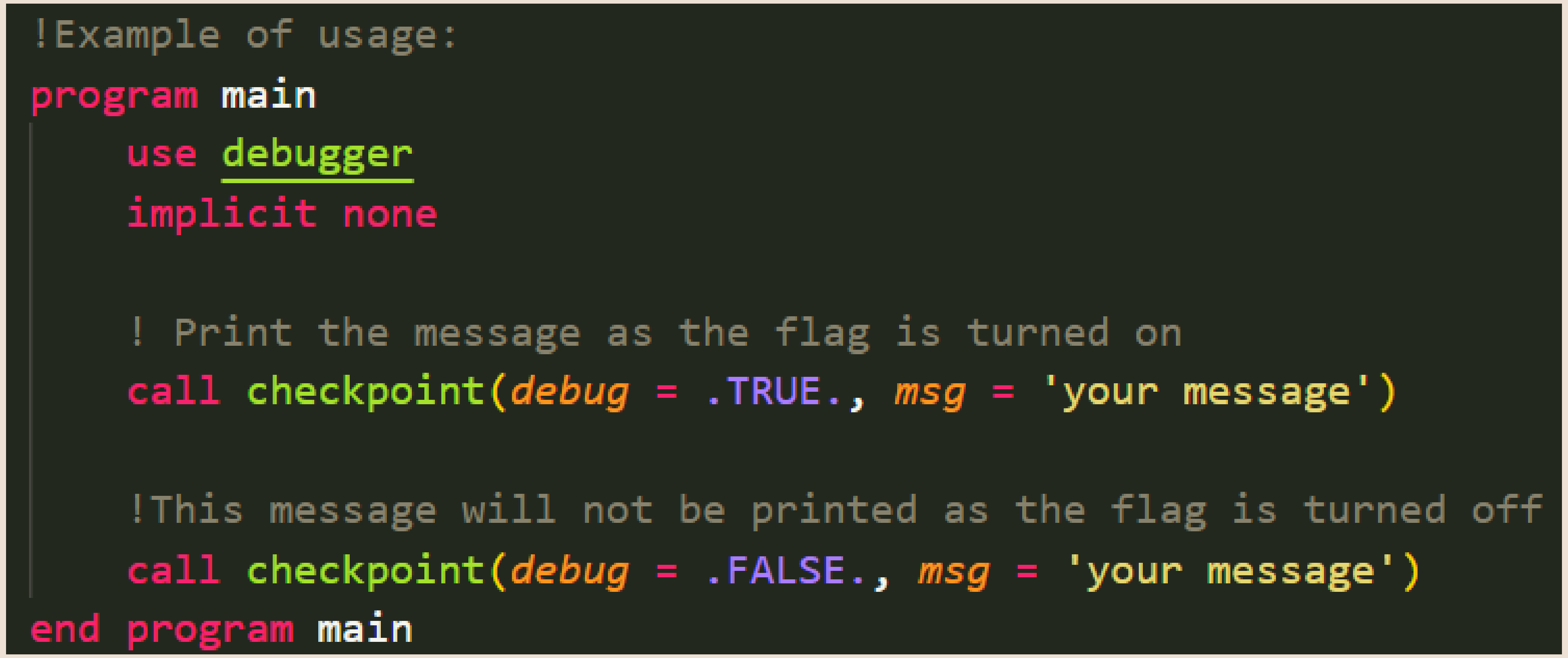
- Improvement of the code from ‘Assignment 1’
  - Debugging module
  - Better commenting and new functions
  - Documentation

- User-defined-type in Fortran
  - Complex matrix in double precision
  - Ad-hoc methods

Debugging modules in Fortran

- Debugging is a ubiquitous and essential part of coding, whatever the language used.
- One of the easiest way of debugging: print(“something”) -> you get the line where the execution ended.
- A preferable way: debugging module (Fortran)
    - Boolean flag

    - Pre-defined checkpoints

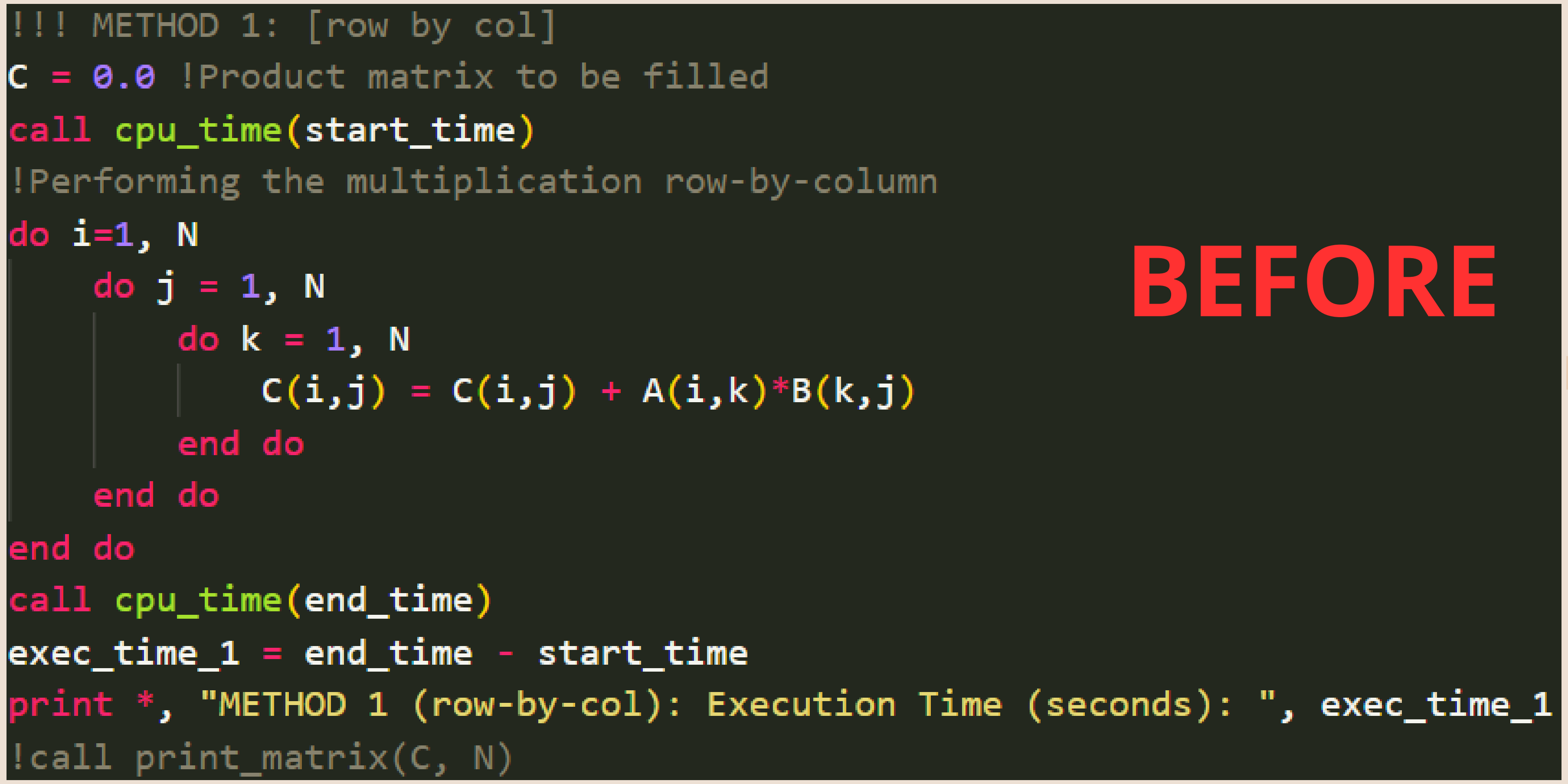


```
!Example of usage:
program main
    use debugger
    implicit none

    ! Print the message as the flag is turned on
    call checkpoint(debug = .TRUE., msg = 'your message')

    !This message will not be printed as the flag is turned off
    call checkpoint(debug = .FALSE., msg = 'your message')
end program main
```

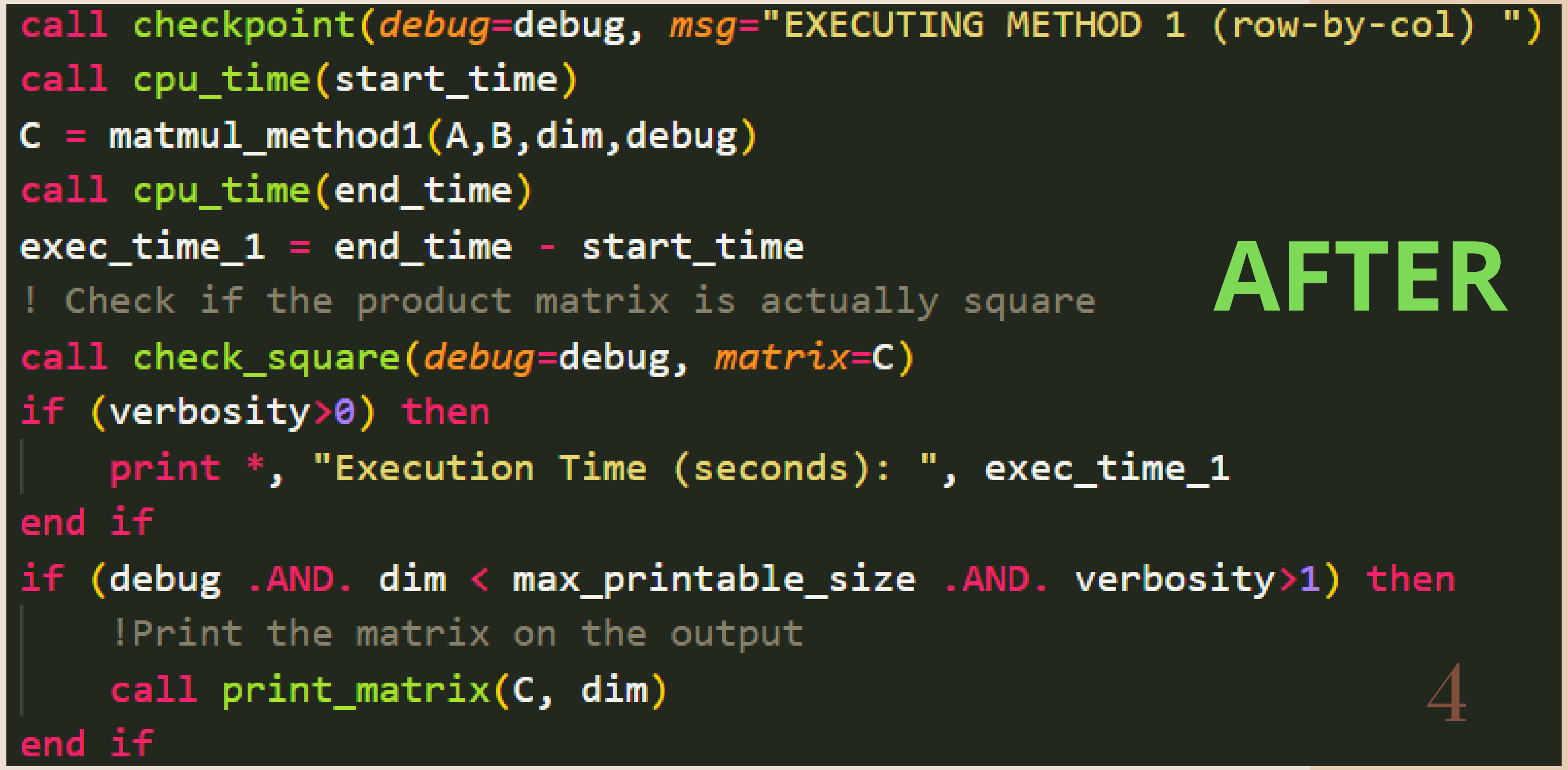
Improvements of assignment 1

Debugging mode and improved readability

- Better **commenting** of the main passages of the code; removed unnecessary comments
- Definition of new **ad-hoc functions** for implementing the specific matrix multiplication methods
- Implementation of **debugging functions** to be executed only in debugging mode: checkpoints, error handling, pre and post conditions (e.g. to check if the resulting matrix is square)



```
!!! METHOD 1: [row by col]
C = 0.0 !Product matrix to be filled
call cpu_time(start_time)
!Performing the multiplication row-by-column
do i=1, N
    do j = 1, N
        do k = 1, N
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end do
    end do
end do
call cpu_time(end_time)
exec_time_1 = end_time - start_time
print *, "METHOD 1 (row-by-col): Execution Time (seconds): ", exec_time_1
!call print_matrix(C, N)
```



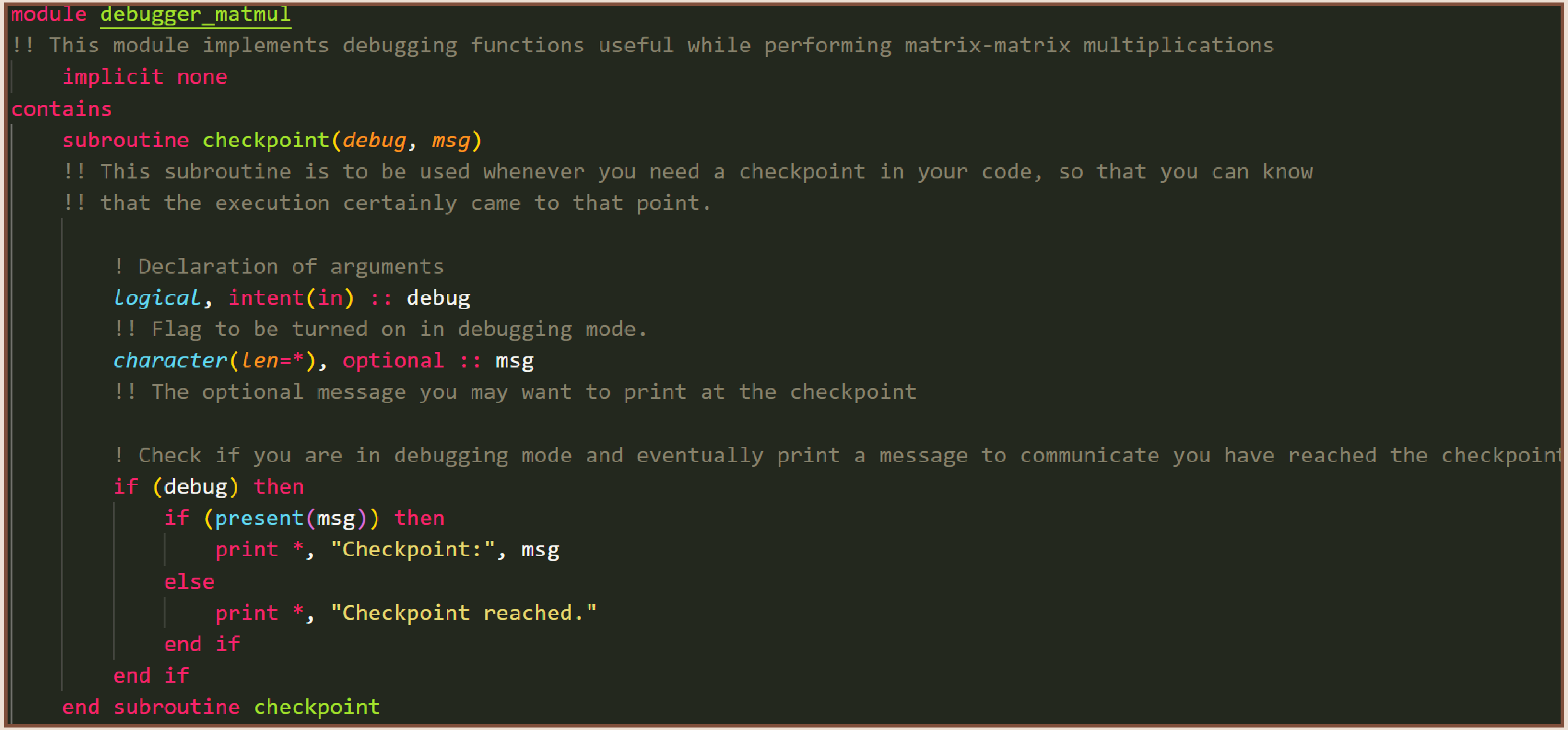
```
call checkpoint(debug=debug, msg="EXECUTING METHOD 1 (row-by-col) ")
call cpu_time(start_time)
C = matmul_method1(A,B,dim,debug)
call cpu_time(end_time)
exec_time_1 = end_time - start_time
! Check if the product matrix is actually square
call check_square(debug=debug, matrix=C)
if (verbosity>0) then
    print *, "Execution Time (seconds): ", exec_time_1
end if
if (debug .AND. dim < max_printable_size .AND. verbosity>1) then
    !Print the matrix on the output
    call print_matrix(C, dim)
end if
```

Improvements of assignment 1

Documentation

Use of **FORD** (FORtran Documenter), an automatic documentation generator for modern Fortran program → produce a documentation informative and nice to look at.

USAGE: Add !! comments in the code to describe functions, variables, and procedures.

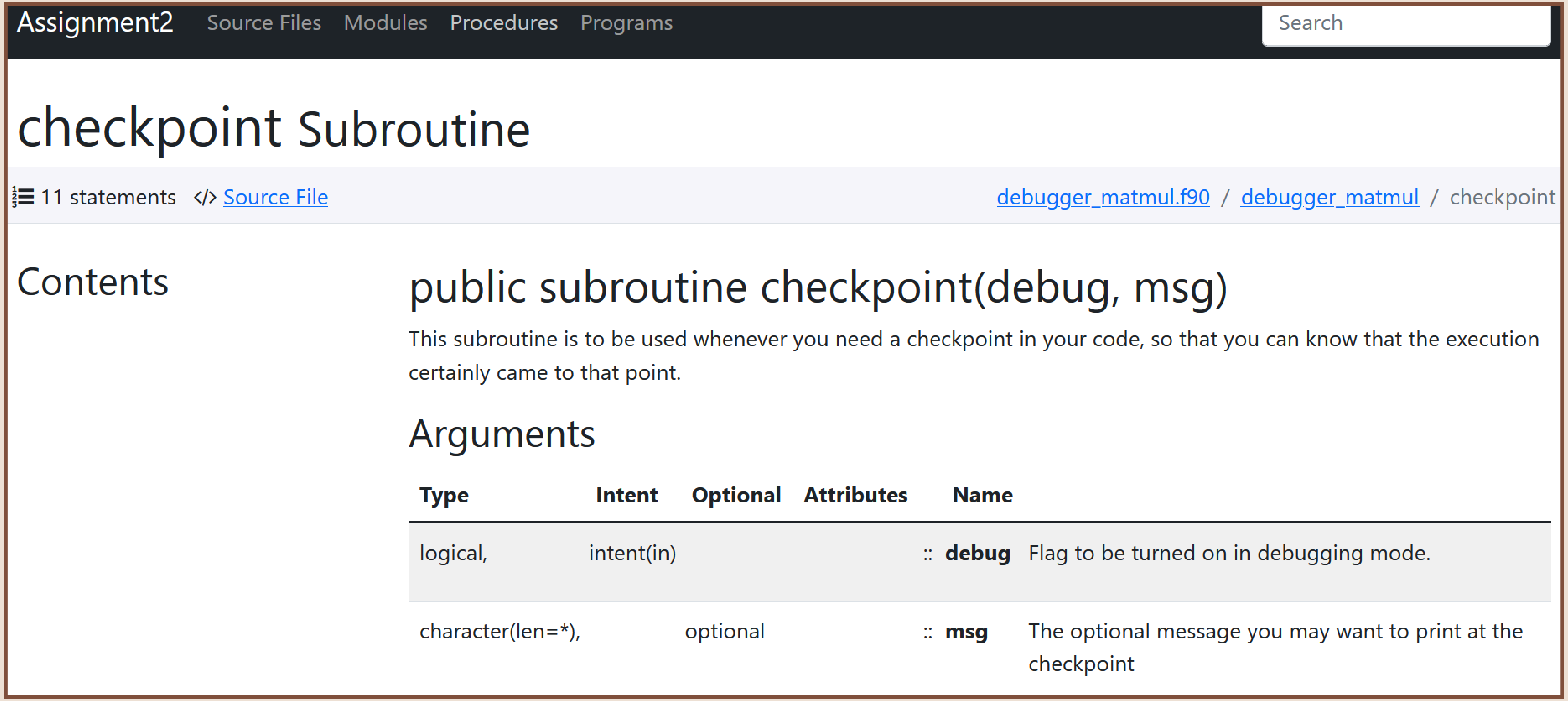


```fortran
module debugger_matmul
!! This module implements debugging functions useful while performing matrix-matrix multiplications
    implicit none
contains
    subroutine checkpoint(debug, msg)
    !! This subroutine is to be used whenever you need a checkpoint in your code, so that you can know
    !! that the execution certainly came to that point.

        ! Declaration of arguments
        logical, intent(in) :: debug
        !! Flag to be turned on in debugging mode.
        character(len=*), optional :: msg
        !! The optional message you may want to print at the checkpoint

        ! Check if you are in debugging mode and eventually print a message to communicate you have reached the checkpoint
        if (debug) then
            if (present(msg)) then
                print *, "Checkpoint:", msg
            else
                print *, "Checkpoint reached."
            end if
        end if
    end subroutine checkpoint
```

Improvements of assignment 1

Documentation

Final result:



checkpoint Subroutine

11 statements </> Source File

debugger_matmul.f90 / debugger_matmul / checkpoint

Contents

public subroutine checkpoint(debug, msg)

This subroutine is to be used whenever you need a checkpoint in your code, so that you can know that the execution certainly came to that point.

Arguments

Type	Intent	Optional	Attributes		Name	
logical,	intent(in)			::	**debug**	Flag to be turned on in debugging mode.
character(len=*),		optional		::	**msg**	The optional message you may want to print at the checkpoint

A type for complex matrices

Necessity of custom structures for storing particular types of data. In Fortran: definition of a **“derived type”.**

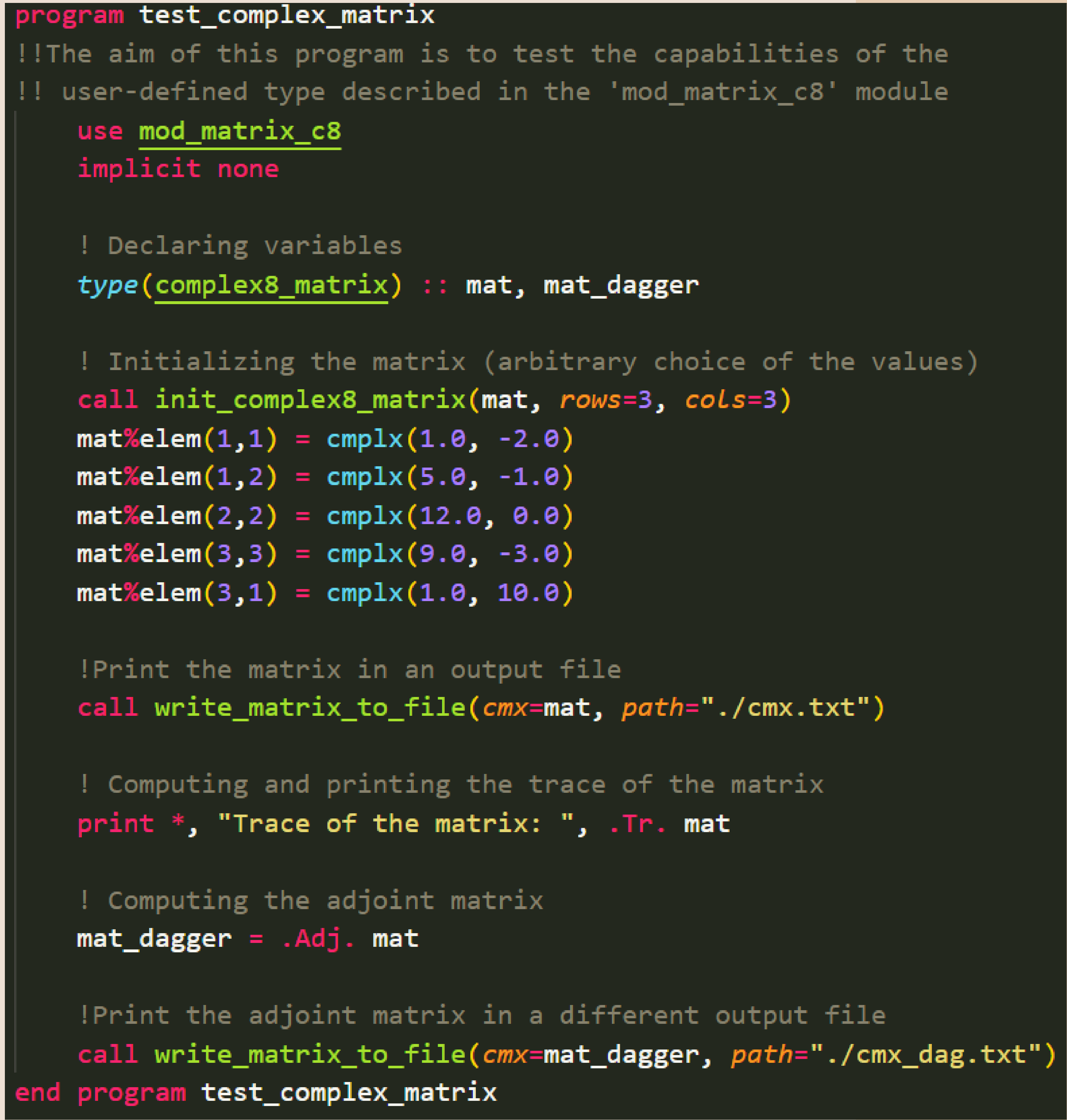
A derived type can have methods (functions) associated with it. It also may be useful to define special operators able to call these methods easily.

An example:

A derived type for dealing with **complex matrices**.

Methods defined:

- Initialization of its elements and storing its size
- Computation of the adjoint matrix + associated operator
- Computation of the trace of the matrix + associated operator
- A function to print the matrix elements onto an external file



```fortran
program test_complex_matrix
!!The aim of this program is to test the capabilities of the
!! user-defined type described in the 'mod_matrix_c8' module
    use mod_matrix_c8
    implicit none

    ! Declaring variables
    type(complex8_matrix) :: mat, mat_dagger

    ! Initializing the matrix (arbitrary choice of the values)
    call init_complex8_matrix(mat, rows=3, cols=3)
    mat%elem(1,1) = cmplx(1.0, -2.0)
    mat%elem(1,2) = cmplx(5.0, -1.0)
    mat%elem(2,2) = cmplx(12.0, 0.0)
    mat%elem(3,3) = cmplx(9.0, -3.0)
    mat%elem(3,1) = cmplx(1.0, 10.0)

    !Print the matrix in an output file
    call write_matrix_to_file(cmx=mat, path="./cmx.txt")

    ! Computing and printing the trace of the matrix
    print *, "Trace of the matrix: ", .Tr. mat

    ! Computing the adjoint matrix
    mat_dagger = .Adj. mat

    !Print the adjoint matrix in a different output file
    call write_matrix_to_file(cmx=mat_dagger, path="./cmx_dag.txt")
end program test_complex_matrix
```

Tha

Thanks for the attention 😊