Debugging and good practices

Abstract

In this assignment, the focus is on the creation of a useful debug subroutine to use in future projects. Then, returning to the third exercise in the very first assignment, some good practices are to be integrated in the code, such as including a documentation, commenting, inserting pre and post conditions, error handling and implementing checkpoints.

Code development

In the first part of the assignment, a `debug` subroutine was implemented. Among the many possible strategies, I chose to create an `INTERFACE` under which to set up many subroutines with different data types but similar structure. Each subroutine accepts 4 variables as input:

- a logical `DEBUG` flag, which is used to actually activate the debug subroutine;
- a generic `var` variable; here 'generic' means that a specific data type corresponds to each specific subroutine;
- a logical `CHECK` flag, which is used in order to perform a check on the variable that is being observed. For instance, for an integer `var` named `x`, such check may be something along the lines of `(x .EQ. 0)`;
- a string `MESSAGE`, which is a short message provided by the user himself.

The main advantage of such a choice is its great flexibility, as a specific instance of subroutine can be readily implemented as the need arises: only the type of `var` needs to be changed. The `INTERFACE` then allows to use the same subroutine name, `DEBUG`, for all the supported data types. The whole module can then be readily used in any project. A template for adding further types is provided at the beginning of the module.

Among the disadvantages of such choice is the fact that the user has to be aware of what needs to be checked: the `DEBUG` subroutine simply prints on screen the answers to the queries provided by the user. An excerpt of the code follows.

```fortran
MODULE DEBUGGER
  IMPLICIT NONE

  ! A module for debugging. For the listed data types, a debug
  ! subroutine is implemented.
  ! - debug : a flag to activate the debug
  ! - var : the variable on which to perform the debug
  ! - check : a logical value, that represents the test to be performed
  ! - message : an output string, provided by the user, which
  !             should describe the eventual error.

  LOGICAL :: debug_on
  !  This is a template for adding debugging options for other data types.
  !  Simply substitute <TYPE> with the desired data type. Do not forget to
  !  add the subroutine to the interface!
  !
  !  SUBROUTINE <TYPE>_DEBUG(debug,var,check,message)
  !    ! a <TYPE> variable debugging subroutine.
  !    ! Calculates the truth value of check and prints
  !    ! the results.
  !    <TYPE> :: var
  !    LOGICAL :: debug,check
  !    CHARACTER(LEN=*) :: message
  !    IF (debug .and. check) THEN
  !       PRINT*, "---------------------"
  !       PRINT*, "__DEBUG INFO__: ", message
  !       PRINT*, "Variable content: ", var
  !       PRINT*, "---------------------"
  !    ENDIF
  !  END SUBROUTINE <TYPE>_DEBUG
  !

  INTERFACE DEBUG
     MODULE PROCEDURE INT4_DEBUG
     MODULE PROCEDURE INT2_DEBUG
     MODULE PROCEDURE REAL4_DEBUG
     MODULE PROCEDURE REAL8_DEBUG
     MODULE PROCEDURE CMPLX8_DEBUG
     MODULE PROCEDURE CMPLX16_DEBUG
     MODULE PROCEDURE CMPLX8MAT_DEBUG
     MODULE PROCEDURE CMPLX16MAT_DEBUG
     MODULE PROCEDURE REAL8MAT_DEBUG
     MODULE PROCEDURE REAL4MAT_DEBUG
     MODULE PROCEDURE INT2VEC_DEBUG
     MODULE PROCEDURE INT4VEC_DEBUG
  END INTERFACE DEBUG


CONTAINS

  SUBROUTINE INT4_DEBUG(debug,var,check,message)
    ! a 4-byte integer variable debugging subroutine.
    ! Calculates the truth value of check and prints
    ! the results.
    INTEGER*4 :: var
    LOGICAL :: debug,check
    CHARACTER(LEN=*) :: message
    IF (debug .and. check) THEN
       PRINT*, "---------------------"
       PRINT*, "__DEBUG INFO__: ", message
       PRINT*, "Variable content: ", var
       PRINT*, "---------------------"
    ENDIF
  END SUBROUTINE INT4_DEBUG
```

The second part was centered around the improvement of an already written piece of code, a module for matrix-matrix multiplication named `MYMATMUL`. Among the possible improvements, the following were adopted[1]:

Documentation: at the beginning of the `MYMATMUL` module, some comment lines briefly describe the contents of the module itself; for every function and subroutine, its purpose and its variables are described.

Comments: in addition to the module documentation, comments were also used in the main program, where they describe the purpose of the variables and outline the various steps, like allocating and filling the matrices. Furthermore, comments are also used to highlight other possible input modes;

Pre- and post- conditions, error handling: a function called `CHECK_DIMS` was written in order to check whether the dimensions of the input matrices are compatible with a matrix-matrix multiplication. This function was used both in the main program, to avoid printing any wrong result altogether, and inside each `MATMUL` function, so that they can have some degree of input control even if not used in this particular context. Being functions, they are bound to return a matrix, and so, even in the case of wrong dimensions, a matrix must be returned. If this happens, the output matrix has the size it should have were the input dimensions correct, and is filled with zeros. A warning is also issued to the user.

The manual input size subroutine `INSERT_DIMENSION` asks the user to insert the dimensions of the matrices; in case of null or negative input, a `DO WHILE` cycle is used to repeat the input insertion, discarding invalid values until a proper one is provided.

The automatic input size subroutine `READ_DIMENSION`, which reads the sizes of the involved matrices from an input file, checks whether the file can be opened; if that is not the case, an error message is printed on screen and the program is closed.

Checkpoints: the `DEBUGGER` module developed in the previous exercise was included in the program, and the `DEBUG` subroutines were used across the program in critical points, such as right after the calculation of the results or right after the allocation of memory, in order to check whether the dimensions were correct. Furthermore, a check was performed to guarantee that all the results from the various algorithms match.

During this check, however, the `DEBUG` subroutine highlighted a mismatch in the results. When running

```fortran
CALL DEBUG(debug_on, 0, (ANY(resu1 .NE. resu2) &
      .OR. ANY(resu2 .NE. resu3) &
      .OR. ANY(resu3 .NE. resu4)),"Wrong!")
```

the output was `Wrong!`, as one of the three comparisons produced at least one non-zero element. Since a previous trial on some smaller (3 by 3) matrices produced four seemingly identical results for all four algorithms, I assumed a rounding error. I then changed the check into a more appropriate

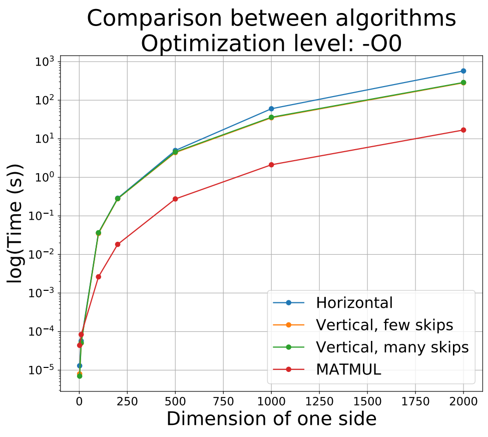
[1]Referenced functions/subroutines not reported here for brevity; they can be found in the annexed code.

```
CALL DEBUG(debug_on, 0, (ANY(resu1-resu2 .gt. 1d-15) &
      .OR. ANY(resu2-resu3 .gt. 1d-15) &
      .OR. ANY(resu3-resu4 .gt. 1d-15)),"Wrong!")
```

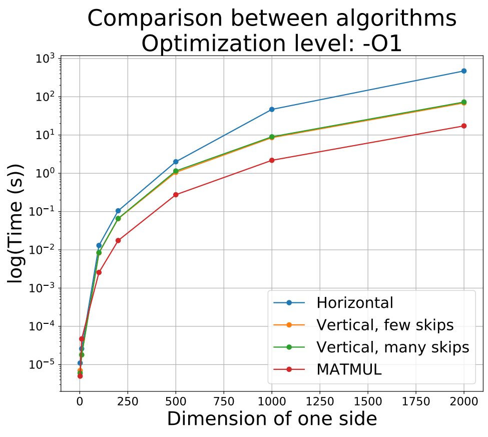
using the precision of a double real as a threshold (as $2^{-53} \sim 1 \times 10^{-15}$). However, this still resulted in an error. Trying then `1d-14`, all checks were finally correct. A direct on-screen print of the difference between results for the four algorithms on (20,20) matrices showed that the intrinsic `MATMUL` differs from the others, with elementwise differences up to (in modulus) $3.55 \times 10^{-15} \simeq 2^{-48}$. For (100,100) matrices, the difference was up to $3.55 \times 10^{-14} \simeq 2^{-44.68}$. Such propagation may affect the results for large dimensions. The discrepancy may be caused by `MATMUL`'s inner workings (such as Strassen algorithm).

In addition, a Python script was implemented that automatically compiles the source file with various optimization flags, creates an input file for the automatic subroutine in the Fortran code and producing square matrices, reads the output of the executables, stores it in a file and then plots it.

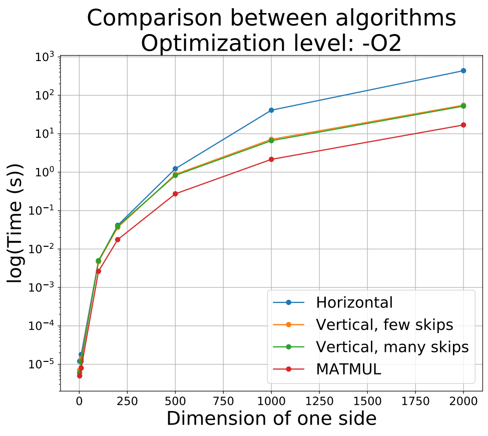
Results



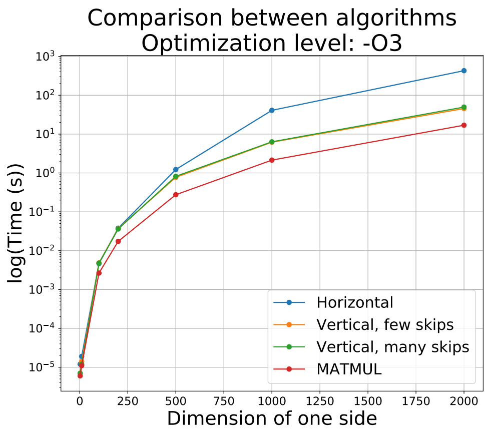
(a) No optimization.



(b) -O1 optimization.



(c) -O2 optimization.



(d) -O3 optimization.

As in week 1, the results show that, for large matrices, the intrinsic `MATMUL` outperforms all the other algorithms, regardless of the optimization. For smaller sizes, instead, performances are comparable. Increasing the optimization level results in a general flattening of the performances, with the 'horizontal' algorithm being the clear worst, and the two 'vertical' ones becoming effectively equivalent once optimized.

Comments and self-evaluation

From the use of the `DEBUG` subroutine, I realized that numerical stability is not guaranteed. It also served as a reminder to use proper conditions when comparing floating point numbers, as finite precision makes equalities prone to errors. Finally, the debugging utilities and the inclusion of good practices served as a way to make my programming more structured and readable.