

## Issues on computation and codes

This chapter summarizes some issues related to programming and how to write programs. Basics of Fortran-90 will also be given for those who have no previous experience in programming beyond Matlab.

### 1.1 Well written programs: Desirable properties

Independently of the chosen programming language, there are several matters that should be considered. It is impossible to write an exhaustive checklist, but the purpose of the short lists below is to bring some of the important issues to the reader's attention. In all programming, one should be careful and pay attention to detail, and remember debug, debug, and debug.

**Portability:** Codes should be written in such a way that they are portable, i.e., they must work correctly in different computer architectures

**Parallelizability:** A good code should be parallelizable. Here, we will not focus on that issue, though.

### 1.2 A few notes on coding and related matters

**Pseudocode:** Write out the details of the algorithm you wish to implement as pseudocode. It makes tracing of errors much easier and programming much faster. Pseudocode is independent of the chosen programming language.

**Modularize:** Write subroutines (=subprograms) for self-contained routines. These subroutines are then called by the main program to perform their tasks. These subroutines may also be tested and verified individually, which is a great advantage.

**Clarity:** In general, try to program in a clear manner avoiding unnecessary complications.

**Variable names:** Use descriptive variable names. It makes your code much for readable. Do not recycle variables which have been named in a descriptive fashion as that will make following the code really hard. You can easily recycle variables such as loop indices ( $i, j, k$ ). It is often a good idea to call temporary variables as *temp* or something like that.

**Comments:** Include generous and precise comments in your code. What does this subroutine do? What are the main variables? Have any tricks been applied? I would recommend to comment blocks of code and avoid line-by-line commenting as it decreases readability.

**Status:** Mark the status of your codes clearly. Is the program, subroutine or function fully functional or tested. How has it been tested? When?

**Print:** Print out results, also intermediate results, as they can be used later for comparison and to help to debug the code.

**Keep a lab-book:** When you program and test your codes. Print out the results, and keep a diary – a lab-book – in which you detail what kind of changes you did and what was the reason for those changes or corrections and their effect. Proper logging will save enormous amount of time afterwards.

- Debug:** Debug, debug, debug, and even after that, debug. It is often quicker to write the first version of a code than to debug it. Even the best of programmers do, however, produce certain number of errors. It has been quoted<sup>1</sup> that the 'industry average' for professional programmers is 15-50 errors per 1,000 lines of code.
- Test cases:** Test your code against known results whenever possible. In particular, pay attention to boundary conditions.
- Program libraries:** Use program libraries when possible. They are tested and verified, and typically optimized for performance and accuracy. Keep in mind, though, that using them is not always straightforward (e.g., wrapping in the case FFTs is sometimes somewhat involved).

### 1.3 A few extra details...

- Summing:** One can minimize roundoff errors if summations are performed in the order of increasing magnitude (add smaller numbers first).
- Avoid mixing of types:** Do not add floating point numbers and integers. Instead, you should use a type conversion (functions such as `REAL` in Fortran) when performing such an operation. `REAL ( i )` converts an integer `i` into a floating point number and it can then be used in operations with other floating point numbers.
- Memory access:** Make sure you access computer memory in the most efficient way. For example, in the case nested loops, the first index runs fastest in Fortran, whereas in C/C++ it is the last one. This can make a huge difference in performance.

### 1.4 Programming

We do not teach programming in this course but here are, nevertheless, some hopefully helpful hints that will get you going if you do not have experience in using programming languages such as Fortran, C/C++, or comparable. Since Numerical Analysis is a prerequisite, you should have experience in using at least Matlab and writing programs that contain loops and `if`-statements. In a nutshell, some of the basic differences between Matlab and real programming languages are the following:

- Matlab is an interpreted language. In simple terms, things are done on the fly and that makes Matlab loops slow. I encourage you to test this by writing simple loops and nested loops using both Matlab and a proper programming language and compare the execution times.
- Programs written in languages such as C/C++ and Fortran must be compiled (this is why you will need to have access to or to be able to install a compiler). This makes such programs much faster.
- Programs written in proper programming languages have much more well defined structure which requires definition of variables and parameters, and they also offer flexibilities in the form of pointers, derived types, allocatable arrays, use of scientific libraries (such as GSL, NAG, BLAS, etc). This is why all high-performance computing in science and engineering is done using programs written in, mostly, C/C++ and Fortran.

Since Fortran happens to be my language of choice, the examples will be given using Fortran. Translation to C/C++ is straightforward.

#### 1.4.1 Do you need a Fortran or a C/C++ compiler?

You can obtain free compilers for different languages from the following web sites. Depending on your system (Linux, Unix, Free BSD, Windows, Mac OSX), you can either obtain a pre-compiled binary or to compile yourself.

- Fortran:** Web: [www.g95.org](http://www.g95.org)  
Web: <http://gcc.gnu.org/wiki/GFortran>
- C/C++:** GCC (GNU) compiler collection.  
Web: <http://gcc.gnu.org/>

<sup>1</sup> "Code Complete" by Steve McConnell

**More:** For Mac OSX, you can obtain compilers using the fink package manager. For Linux, you can also use your package manager (Ubuntu, Fedora, Debian, etc). All of them provide a whole set of compilers and programming tools to choose from. Using a package manager offers a quick way to obtain the necessary tools including the GSL scientific library and many other libraries.

### 1.4.2 Fortran – FOrmula TRANslation

As said above, Fortran is my language of choice and hence I will provide some help to get you going with Fortran. Once you learn one language, it is easy to learn another. The important concepts are the same in more or less all languages, it is only the syntax that varies. As for execution speed, Fortran and C have typically been the fastest ones and hence the preferred languages in high-performance computing and simulations. C++ has also gained a lot of popularity over the past years.

Fortran is one of the oldest high-level programming languages (first published in 1957). It is easy to learn and it is very efficient; after all, it was made for number crunching.

#### Basic elements of Fortran

Here we list some of the basic elements of Fortran-90. The sample programs are available at the course web site for you to try and modify them yourself.

The basic program structure in F90 looks like this:

```
PROGRAM    program_name
  IMPLICIT NONE
  [specifications]
  [execution]
END PROGRAM program_name
```

- Start:** The main program starts with the keyword `PROGRAM` followed by the name of the program.
- End:** The main program finishes with the keywords `END PROGRAM` followed by the name of the program
- IMPLICIT NONE:** This is very important to have. It means that the implicit Fortran naming conventions are not used but all variables must be declared explicitly
- specifications:** This is where variable declarations are done
- execution:** This is the part where the execution of the commands takes place

Note: Fortran-90 has free format, i.e., the empty spaces do not matter. Fortran 77, on the other hand has fixed format; the first seven characters of each line must be blank (except for possible line number and/or continuation character).

#### Comments

In F90 comments are started with the character `!`. They may cover an entire line or just a part of a line (in f77 only full lines can be commented). The exception is the case when `'!` is part of a character string. Blank lines are also interpreted as comments. For example:

```
!-----
!  
! Write something on screen  
!  
!-----
!  
PROGRAM    SIMPLE  
  IMPLICIT NONE           !- The important IMPLICIT NONE statement  
  write(6,*) 'Write this on screen'  
  write(6,*) 'Notice: Inside character string ! does not start a comment.'  
END PROGRAM SIMPLE
```

Let's compile and execute this masterpiece (here you also see how to compile programs under Linux/Unix/Mac OSX using gfortran; gfortran is freely available):

```
bash-3.2$ gfortran simple.f90 -o simple.out
bash-3.2$ ./simple.out
Write this on screen
Notice: Inside character string ! does not start a comment.
bash-3.2$
```

### 1.4.3 Logical expressions

```
equal: == or .eq.
not equal: /= or .ne.
greater than: > or .gt.
less than: < or .lt.
greater than or equal: >= or .ge.
less than or equal: <= or .le.
```

### Reading and writing

The above program clearly does the job it is indented to do. The above program introduced also the most basic F90 I/O: writing on screen. The WRITE command is of form WRITE ([number], \*). The number defines the channel for output and the star states that writing is done using free format; formatting can also specified, but we will get to that later. Channel number 6 is reserved for writing on screen, and channel number 5 for reading from screen. The example below shows also how to write into a file using Fortran's generic output method. The default name for output file is then `fort.channel_number`. Fortran has a lot of formatting and I/O options, please see a Fortran manual for them. Notice: reading from an external file (using the default name `fort.channel_number` works the same way, for example, READ(10,\*) `iii` would read variable `iii` from file `fort.10`). For more on I/O, please see a Fortran manual.

```
!-----
!  
! Write an integer on screen,  
! read it, and write on screen.  
!  
!-----
!  
PROGRAM SIMPLE  
  IMPLICIT NONE                                !- The important IMPLICIT NONE statement  
  
  Integer :: i  
  
!-- Channel 6 is reserved for writing on screen and  
!-- channel 5 for reading from screen  
  
  write(6,*) 'Please give an integer:'  
  read(5,*) i  
  write(6,*) 'This is the integer you gave: ', i  
  write(6,*) 'It was also written in file fort.10'  
  
!-- The rest of the channels are ok. If used in the most  
!-- basic form as below, output goes to file fort.channel_number  
!-- It is also a good idea to close the channel.  
!-- See fortran manual for more involved I/O.
```

```

write(10,*)'# Output from program simple2.f90'
write(10,*)' '
write(10,*)i
close(10)

```

```
END PROGRAM SIMPLE
```

And this is how it functions:

```

bash-3.2$ gfortran simple2.f90 -o simple2.out
bash-3.2$ ./simple2.out
Please give an integer:
56
This is the integer you gave:          56
bash-3.2$

```

### Declaring variables and parameters

There are a few rules which should be noticed when declaring variables and parameters.

- In contrast to C/C++, all identifiers in Fortran are case *insensitive*
- An identifier *must* start by a letter. Digits and underscores are allowed elsewhere
- The length of an identifier is limited to 31.
- Important: Fortran does not have reserved words. That means that you can use words such as INTEGER as variables. That should, however, be avoided as it may lead to lots of problems.
- Fortran has implicit type assignments. If IMPLICIT NONE is not used, then the compiler assumes all variables beginning with letters (both upper and lower case as Fortran does not recognize cases) I–N as integers, and the rest as reals. This can get very confusing and lead to a lot of errors. IMPLICIT NONE should always be used.

The difference between a variable and a parameter is that you must assign a value to a parameter at the time of its declaration and that value cannot be changed afterwards; upon compilation, the compiler 'freezes' its value. Variables, as their name suggests, may change.

The following are the intrinsic variable types in F90.

**INTEGER:** Example:

```

INTEGER :: i,j,k !- dummy variables for counters, etc.
INTEGER, PARAMETER :: Maximum=100 !- integer parameter

```

**REAL:** Example:

```

REAL :: temp !- temporary floating point variable
REAL, PARAMETER :: PI = 3.141592 !- real parameter

```

**COMPLEX:** Complex numbers.

**LOGICAL:** Can be either true or false.

**CHARACTER:** Example:

```

CHARACTER(LEN=20) :: LastName, FirstName !- string can be 20 characters long
CHARACTER(LEN=*) :: LastName, FirstName !- length is determined by compiler

```

Notice: You can initialize a variable by giving its value when declaring it. Sometimes it makes more sense to initialize them later. Example:

```

INTEGER :: i=1,j=1,k=1 !-- dummy variables for counters, etc.}

```

**Loops and if statements & arrays**

```

=====

PROGRAM LOOPING

=====
! This program demonstrates how
! simple loops work
=====

    IMPLICIT NONE
    INTEGER :: i, j

!-- simple loop

    do i=1,3
        write(6,*) ' i= ',i
    enddo

!-- double loop (also called nested loop)

    do i=1,3
        do j=1,2
            write(6,*) ' j= ',j, ' i= ',i
        enddo
    enddo

!-- general do-loop with exit statement
!-- without the if statement paired with exit
!-- this would be an infinite loop.
!-- Notice also that the initial value of variable i
!-- is the one that is left after the execution of the above loop

    do
        write(6,*)i
        i=1+i
        if(i > 10) exit
    enddo

END PROGRAM LOOPING

=====

=====

PROGRAM MORELOOPING

=====
! This program demonstrates how
! simple loops work
=====

    IMPLICIT NONE

```

```

INTEGER :: i, j
INTEGER, DIMENSION(10) :: myvector !- vector of length 10
INTEGER, DIMENSION(2,2) :: myarray !- 3x3 array

!-- simple loop

do i=1,10
  myvector(i) = 1+i
  write(6,*) myvector(i)
enddo

!-- double loop (also called nested loop)
!-- IMPORTANT: in fortran the first index runs fastest
!-- in C/C++ it is the second one.

do i=1,2
  do j=1,2
    myarray(j,i)=i+i*j
    write(6,*) j,i,myarray(j,i)
  enddo
enddo

!-- an example of using if statements (very artificial example)
!-- and comparing values using .AND. and .OR.

do i=1,10
  do j=1,10
    if (i > 2 .OR. j>2) then
      if(i > 2 .AND. j>2) then
        write(6,*)'both i and j are too large', j,i
      else if(i>2 .AND. j <= 2) then
        write(6,*)'i is too large', i
      else if(i <= 2 .AND. j > 2) then
        write(6,*)'j is too large', j
      endif
    else
      myarray(j,i)=i+i*j
    end if

  enddo
enddo

END PROGRAM MORELOOPING

!=====

```

### Subroutines and functions

Fortran has two kinds of subprograms, *functions* and *subroutines*. The practical difference is the following: a function returns only one value, no more, no less. A function can take more arguments, but it returns exactly one value.

Subroutines may return an arbitrary number of values. The syntax is very similar to the main program as discussed above:

```
SUBROUTINE    subroutine_name (argument list)
  IMPLICIT   NONE
  [specifications]
  [execution]
END PROGRAM  subroutine_name
```

There are elaborations to subroutines but this should be enough to get you going. It is a good idea to use subroutines to separate operations that are well defined and which should not be part of the main code. For example, if you need to solve a Laplacian in your program, it is a good idea to write a subroutine for the Laplacian and call it from the main program.

Subroutines may be either external or internal. The easiest way in short programs is to keep them as part of the code.