Coding Beautifully

Jan Tomczak
Moritz Siegel

March 2021

1 Debug flags

Once you have implemented an algorithm, the program might not do what you intended, crash, or not even compile. Often, such problems arise from very minor mistakes that can, however, be hard to spot. In many cases, the compiler can provide detailed information about what goes wrong. However, the compiler needs to be told to do so. Indeed, keeping track of your variables etc slows down the execution of your program, so this is turned off by default.

The following compiler options will help you debug your program!

1.1 Compiling Fortran

To compile your program for debugging purposes, use

```
gfortran FLAGS your_prog.f90
```

with the following FLAGS

`-Og`
"optimize debugging experience (O for optimise)"

`-Wall`
"This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid"

`-fimplicit-none`
please always use the "implicit none" option in the main program, as well as procedures and functions, e.g.,

```
program my_stuff
implicit none
integer i
...
```

activating the `-fimplicit-none` flag will complain, e.g., if a variable `j` is being used without its type (integer) being declared in the header. This helps, for instance, not to mix up integer and float variables.

`-fcheck=all`
This checks, for example, whether the index `i` of an array `vector(1:n)` is out-of-bound `i>n` or `i<1`.

`-fbacktrace`
"When a serious runtime error is encountered or a deadly signal is emitted (segmentation fault, illegal instruction, bus error, floating-point exception, and the other POSIX signals that have the action 'core'), the Fortran runtime library tries to output a backtrace of the error" ... so the compiler also outputs information from before a crash.

For production runs, i.e. your program works and you want to obtain final data/results, omit all of the above (the program will be faster).

2 Filename Extensions

It is common practice to assign the ending `.f90` to all *modern* Fortran source files, even if they are written with a later standard (e.g. 2008). The main goal is to differentiate between Fortran77's `myprogram.f` and Fortran90's `myprogram.f90`.

3 Automating: Compile + Run + Plot

To save yourself having to compile your program manually every time you change something, then executing it, and finally being able to plot the results; all this can also be summarized in a shell script `myprogram.sh`:

```
#!/bin/sh
time gfortran -o myprogram myprogram.f90 &&
time ./myprogram &&
time gnuplot myprogram.gp
```

where the `#!/bin/sh` tells the operating system (some Linux of course), what program to use to run the script; shell `sh` in this case). The `&&` ensure that each command is only executed, if the preceding one did not give any errors. The `time` command logs the computation time one can use to roughly estimate computation speed.

This shell script `myprogram.sh` can be run in any terminal (using the shell of ones choice e.g. bash) calling:

```
bash myprogram.sh
```

The `myprogram.gp` stated in the shell script also is a generic text file filled with gnuplot commands, for example:

```
#!/bin/gnuplot
reset
set key
set size square
set term png size 1920,1080
set out 'myprogram.png'
set title
set xlabel 'x'
set ylabel 'y'
p 'myprogram.txt' u 1:2 w p pt 7 lc 0 ps 2 t 'data', \
...
```

where the `p` stands for *plot*, as any command in gnuplot can be shortened to the last distinct letter (which is found by trial and error).

4 Variables

4.1 Precision

A general and important topic in Fortran is precision. Real and double precision are not interpreted in the same way by different compilers, so to be on the safe side, it is recommended to initialise variables with a fixed precision:

```
program my_stuff
implicit none
integer, parameter :: p=selected_real_kind(16), xint = 8
real(kind=p) :: yreal = 1.23_p
```

for *casting*, conversion within the program the safest option would be:

```
n = real( xint, kind=p )
n = real( 1.23_p, kind=p )
```

4.2 Always Initialise Counters with Zero

more important - especially in Fortran - counters must be initialised `n = 0` otherwise the following happens when a loop first increments the counter `n = n + 1`: The Fortran compiler usually does not check whether the variable n has already been assigned a value, it just takes what is in RAM at the location of this variable— which can be anything—and adds `1` to it!

4.3 About Pi

If needed, it is convenient to define π the following way in the preamble:

```
double precision, parameter :: pi = acos(-1.d0)
```

or even better:

```
real(kind=p), parameter :: pi = acos(-1._p)
```

4.4 Integer Overflow

Using a *normal* integer, the highest value possible is `imax = 32767`. If the value is raised further by e.g. as a running variable inside a loop, the compiler usually doesn't notice, then the value suddenly jumps to: `imax+1 = -32767`.

4.5 Vector Operations

In Fortran, there is no built-in way of multiplying matrices (including vectors), nor for dot-product or even cross product! Basic arithmetic operations $\odot$ like addition, subtraction, multiplication or division, etc., on variables are always done element-wise:

$$\texttt{c = a} \odot \texttt{b} \rightarrow c_n = a_n \odot b_n \quad \forall n$$

4.6 Exponentiation

Due to intrinsic hardware-level operations, the notation `x**2` is usually faster than `x*x`; with `x**2.0` being the slowest (if the compiler doesn't realize 2.0 actually represents a sort of *hidden* integer).

4.7 Correct rounding

Usually when converting from a real number to an integer you want to use `nint(x)` instead of `int(x)`. `int` simply cuts off all decimal places, while `nint` rounds to the nearest whole number. When using `int`, small rounding errors can lead to unexpected behaviour. E.g.:

```
real :: pi = acos(-1.0)
real :: x, y
integer i

x = 60*pi
y = 5*pi
i = int(x / y)    ! x is now 11 !!!
i = x / y         ! same as int(x), x is now 11
i = nint(x / y)   ! x is now 12
```

4.8 Integer Division

When dividing two integer variables the result is converted to an integer by removing all decimal places, even if the result of the calculation is assigned to a real variable!

```
integer :: i = 1, j = 2
real :: x

x = i / j
! x is now 0.0
```

In order to get floating point division, at least one of the variables has to be converted to real:

```
integer :: i = 1, j = 2
real :: x

x = real(i) / j
! x is now 0.5
```

4.9 Loops with real variables

The most precise form to step through an interval with `n` intervals is the following form:

```
real :: begin, end, x
integer :: n, i

do i = 0,n
    x = begin + (i*(end-begin))/n
end
```

Avoid incrementing x inside the loop (`x = x + dx`) because this way small rounding errors of dx can sum up.