1 Worksheet 5: Benchmarking Fortran Routines

In this worksheet you will use the previously developed module `Erf_Series` for calculating $\mathrm{erf}(x)$ via a Taylor series. You will probably find it useful to also reuse pieces of the your code from worksheet 4. In this section we will learn how to time the execution time for a Fortran routine such as `ErfOfReal`. Also, since our project is already getting big and we are reusing a lot of code, you will learn how to compile modules separately and then link them. Along the way, you will hopefully learn something about compiler optimization switches as well.

You will need to read section 2.7.2 from the manual and review all the previous material until you understand it. The next worksheet will introduce arrays for the first time and we will use them to plot the results from this worksheet using a Fortran graphics library called DISLIN.

1.0.1 Fortran 90 timing subroutine `SYSTEM_CLOCK`

The Fortran manual you got does not discuss this routine, but it is a simple and very useful routine. The syntax of this subroutine is,

`SYSTEM_CLOCK([COUNT=clock_count],[COUNT_RATE=clock_rate],...)`

where both the `COUNT` and the `COUNT_RATE` arguments are optional. Each computer and compiler have their own fast clock. This clock is ticking from the start of the routine at a given number of ticks per second. The value of this counting rate is returned in the integer `clock_rate`, while the current value of the clock counter is returned in the integer `clock_count`.

This will become clear via an example. Let's assume you want to calculate the time it takes for a given action to complete. Then you would use:

```
INTEGER :: clock_start,clock_end,clock_rate
REAL(KIND=sp) :: elapsed_time
...
CALL SYSTEM_CLOCK(COUNT_RATE=clock_rate) ! Find the rate
CALL SYSTEM_CLOCK(COUNT=clock_start) ! Start timing
   ...Do your calculation here, for example:...
   ...erf=ErfOfReal(x)...
CALL SYSTEM_CLOCK(COUNT=clock_end) ! Stop timing
! Calculate the elapsed time in seconds:
```

```
elapsed_time=REAL((clock_end-clock_start)/clock_rate,sp)
```

1.1 Timing the `Erf` function

One of the difficulties of timing a rouine like `Erf` is that it is very fast—of the order of several microseconds (μs). Most clocks can only resolve milliseconds or tens of milliseconds. The basic strategy in this case is to call the routine `n_repetitions` times in a `DO` loop and time the total time for the execution.

Write a new module called, for example, `Erf_Timing`, that will contain a single routine inside, say `ErfTiming`, which will take x as an argument and return the time (in seconds) it took for the routine to execute (per instance). Put this module in a *separate file*.

Your function may look something like (by now you should know how to place this in a module and properly declare the arguments):

```
FUNCTION ErfTiming(x) RESULT(elapsed_time)
  ...
  CALL SYSTEM_CLOCK(COUNT=clock_start)
  DO i=1,n_repetitions
    erf=ErfSeries(x)
  END DO
  CALL SYSTEM_CLOCK(COUNT=clock_end)
  ...Return the elapsed time in us...
  ...Remember to divide by n_repetitions!...
END FUNCTION ErfTiming
```

Play with the value of `n_repetitions`. Reasonable values are anywhere from 10,000 to 1,000,000. You will know whether the value is large enough if the results of your program do not fluctuate when you execute the program several times. You will know it is too large if you have to wait for more than a few minutes for your program to complete.

1.2 Advanced: Compiler optimization and timing

The above will work OK in most cases, but it has some dangers that need to be pointed out. Namely, smart compilers will see that the body of the `DO` loop above does not change and so will not execute the loop many times but only once. This falls under the great Fortran strength of compiler optimization and is discussed in section 2.7.2 of the manual. This is especially likely in

Fortran 90, where the compiler can be informed (say by the `PURE` attribute) that the `Erf` routine is "harmless"–it has no outside effects.

To ammend this, a common strategy is to turn compiler optimization off when compiling *only the timing routine* (`ErfTiming`). On many compilers this is done by adding a `-O0` switch when compiling (in words, set optimization level to 0). For example:

```
>   f90-vast -O0 timing.f90 ...
```

1.3 The main program

Last time you wrote a program in which you calculated the error function for `n_points` values in the interval [`x_min`,`x_max`] and wrote it to a file. This time just remove the file I/O statements and replace the call,

```
erf=ErfOfReal(x) ! Or erf=ErfSeries(x)
```

with

```
elapsed_time=ErfTiming(x)
```

Then print the value of x and the elapsed time *in microseconds* (of course, of you want to, you can simply write these to a file and do as least modifications as possible). Again, let the user enter the number of output points `n_points`, `x_min` and `x_max`, and the precision ε. For example, the output of your code may look like:

```
[donev@gauss erf]$ ./erf_timing.x
Enter: x_min, x_max, n_points and error: 0.0,4.5,5,1E-6
x=     0.0000     time =     1.0000 us
x=     1.1250     time =     4.5000     us
x=     2.2500     time =     7.7000     us
x=     3.3750     time =     14.600     us
x=     4.5000     time =     22.500     us
```