Exercise n. I
Basics

1. **Overflow and underflow**
   In order to investigate which are (within a factor of 2) the **overflow** (the greatest number that can be stored) and **underflow** (the smallest) limits, you can write a code doing something like that (*note: this is a pseudocode to have an idea of the algorithm, it is not written in a precise language*):

```
under = 1.
over = 1.
   do until.... (or: do N times, with N =...)
   under = under/2.
   over = over * 2.
   write: number of iteration, over, under
   end of cycle
```

   If you want, you can use the available codes (where `r`=real, `s`=single precision, `d`=double precision) which can be compiled with `gfortran` (or `g95`, `F`, `fort`, or other fortran compilers).

   (a) Check overflow and underflow for *floating point* numbers in single precision. (see `rs_under_over.f90`)

   (b) Do the same in double precision. (see `rd_under_over.f90`)

   (c) Do the same for integers (Hint: to be more precise, consider also the numbers obtained by multiplying times 2 and subtracting 1...) (see `i_min_max.f90`):

   (d) *(Optional)* Some compilers convert "underflow" with "0"; same are able to handle exceptions... If you have other fortran compilers installed, compare what you obtain in (a)–(c) using different compilers. (For instance, if you use `F` instead of `g95`: use `F` without/with the option `-ieee=full` (for exception handling): `F -o test.out -ieee=full`. What do you get by compiling the code with/without the option and running again?)

2. **Machine precision**
   Write a program to determine the **machine precision** ε (i.e. the smallest positive number that -added to the unit- does change its value stored in memory). For instance you could do something like that (*pseudocode*):

```
eps = 1.
   do until.... (or: do N times, with N =...)
   eps = eps/2.
   uno = 1. + eps
   write: number of iteration, over, under
   end of cycle
```

(a) Check the machine precision for *floating point* in single precision. (see `rs_limit.f90`)

(b) Do the same in double precision. (see `rd_limit.f90`)

(c) Check your results calling the *intrinsic function* `epsilon()` (see `strano.f90` and `d_strano.f90`).

3. **Good and bad algorithms, truncation and roundoff**
A typical numerical problem is to calculate a function for a given value of a variable as the sum of a series. For instance:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

(a) Write a program to calculate in single precision e^{-x} as the sum of the series above, with an absolute error that you choose, and save the results in a table like this:

```
x    i(no. of terms of sum)    sum    |sum-exp(-x)|/exp(-x)
```

where `sum` is the sum of the first `i` terms of the series and `exp(-x)` is calculated with the intrinsic function, and it can be therefore considered as the value of the infinite series, so that `|sum-exp(-x)|/exp(-x)` is the relative error.
As an exercise, you could write and compare different codes:

i. Using the *factorial* function (see `test_factorial.f90` for the use of a *recursive* function). Make some tests fixing x but changing the number of terms of the series, checking if (and up to which term) the factorial is correctly calculated.

ii. Avoiding the use of the factorial. *You have an example of code avoiding the factorial: (*`exp-good.f90`*. It also avoids odd powers of x, and does a smart use of the previous terms.*

Which program works better?

(b) Consider the best code. Use it for small and large x, for instance x=0.1, 1, 10, 100, 1000, and consider the results obtained. In particular: what about overflow o underflow? Change the code to calculate $e^{-x} = 1/e^{x}$ and not directly the series above. Is it better? Why?

(c) Consider the most efficient way to calculate e^{-x} as a series of negative and positive terms; change the code using the double precision. Compile, run, and comment on the results.

4. **Roundoff: derivative**

   - Write a code (e.g., see `deriv.f90`) to calculate the derivative of $f(x) = sin(x)$ in $x = 1$ with the formulas:
     - 3-point symmetric: $f'(x) \sim \dfrac{f_1 - f_{-1}}{2h}$
     - 2-point "forward": $f'(x) \sim \dfrac{f_1 - f_0}{h}$
     - 2-point "backwards": $f'(x) \sim \dfrac{f_0 - f_{-1}}{h}$

     where $f_0 = f(x)$, $f_1 = f(x+h)$, e $f_{-1} = f(x-h)$.
   - Use $h = 0.5$, 0.2, 0.1, then $h/10$, $h/100$, $h/1000$, $h/10000$, and reports the results in a table to compare the three algorithms. It's more convenient to report the error ('calculated−exact' value, since in this case we know the exact value...)
   - Comment the results. What about roundoff errors?

```
!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
! calculates the factorial using a recursive function; use of module
!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

module fact
public :: f
contains

recursive function f(n) result (factorial_result)
   integer, intent (in) :: n
   integer :: factorial_result

   if (n <= 0) then
      factorial_result = 1
   else
      factorial_result = n*f(n-1)
   end if
end function f

end module fact

program test_factorial
   use fact

   integer :: n
   print *, "integer n?"
   read *, n
   print "(i4, a, i10)", n, "! = ", f(n)
end program test_factorial
```

```
!ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
!     exp-good.f : a GOOD ALGORITHM to calculate e^-x
!                  as a FINITE sum of a series
!                  (to compare with exp-bad.f
!                   and with the machine intrinsic function)
!ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 program expgood
 !
  ! variable declaration:
  !     x
  !     accuracy limit: min
  !
  implicit none
  real ::  element, sum, x, min = 1.e-10
  integer :: n
  open(unit=7,file="exp-good.dat",position="append",action="write")
  write(unit=7,fmt=*) "x, n, sum, exp(-x), abs(sum-exp(-x))/sum"
  !
  ! execute
  !
  write(*,*)' enter x:'
  read(*,*) x
  sum     = 1
  element = 1
  do  n=1, 10000
     element = element*(-x)/n
     sum = sum + element
     if((abs(element/sum) < min) .and. (sum /= 0)) then
        write(*,*) x, n, sum, exp(-x), abs(sum-exp(-x))/sum
        write(unit=7,fmt=*) x, n, sum, exp(-x), abs(sum-exp(-x))/sum
        go to 10
     endif
  enddo
10 continue
  close(7)
  !      stop "data saved in exp-good.dat"
end program expgood
```

```
!ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
        program deriv
!
!  numerical derivative: left, right, symmetric in SINGLE PRECISION
!
!ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      real :: h(8)
      real :: x, exact
      integer :: i, N=8
      data h/0.5, 0.2, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001 /
!
      print*, " h, f'_ds, error, f'_sin, error, f'_simm, error "
      x = 1.0                   ! inizialize variables
      exact = cos(x)
       do i=1,N
      deriv_ds = (sin(x+h(i))-sin(x)) / h(i)
      deriv_sin = (sin(x)-sin(x-h(i))) / h(i)
      deriv_simm = (sin(x+h(i))-sin(x-h(i))) / (2*h(i))
         print*,  h(i), deriv_ds, deriv_ds - exact, deriv_sin, deriv_sin - exact, &
  & deriv_simm, deriv_simm - exact
      end do
      stop
      end program deriv
```

A few notes on these exercises:

- **"do loops":**

```
do i=1,n
...(i)...
...
end do
```

  or *"named do"*:

```
myloop : do
...
end do myloop
```

  Note the condition to exit from a loop:

```
do i=1,n
  if (...) then
  ...
  exit
  ...
end do
```

  interrupts the loop, which is the same of (older style):

```
      do i=1,n
        if (...) then
        ...
        go to 10
        ...
      end do
 10   continue
```

  whereas:

```
do i=1,n
  if (...) then
  ...
  cycle
  ...
end do
```

  go to the next value of `i` (skipping lines after `cycle`) and continues the loop.

- open/close files (remeber: default reading/writing units: 5/6)
- unformatted output (`print*` or `write(...,fmt=*)`)
- variable and type declarations (better to use `implicit none`+...)