Technisch Wetenschappelijke Software
Scientific Software
Session 2: Fortran 95, Working with numbers on a digital computer

"Real numbers do not exist in a computer."

Introduction

This exercise session examines some peculiarities encountered when working with numbers (both integer and real) on a digital computer. The aim of this session is for you to develop an understanding of the following concepts:

- integer overflow;
- the floating point kind types available in **Fortran 95** and their properties;
- floating point issues, such as catastrophic cancellation, the effect of rounding errors and overflow.

The syntax you need for the session can be found in the preparatory reading, but you can ask further questions to the assistants in the session. If you have questions later, please post them on the Toledo discussion forum. That way other students can see the questions that have already been asked, as well as their answer.
Questions preceded by *Extra* should only be solved during the session if time permits, otherwise try to finish them at home.

Exercises

Question 1: The double factorial function

Link preparation: Scientific Software Development with Fortran - 2.7 (Number Kinds and Precision), 2.12 (Conditionals); Fortran 2003 standard - NOTE 12.38 (An example of a recursive function), 13.5.4 (Kind functions) and their specifications in 13.7.

Last week we encountered `elemental` procedures, which can be used to apply an operation element-wise to an array. This week we will use `recursive` functions, which are functions that call themselves[1]. A function can be declared recursive by adding the keyword `recursive` before the specifier `function`.

```
recursive function double_factorial(..) result(doublefact)
...
end function double_factorial
```

Important: for recursive functions the name of the output variable should be specified using a `result` clause and its name should differ from the function name. *Extra:* Can you see why?

Write a `recursive` function that calculates the double factorial of an **integer** n. The double factorial of n, denoted by $n!!$, can be computed recursively:

$$n!! = \begin{cases} n \cdot (n-2)!! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \text{ or } n = -1 \\ 0 & \text{if } n < -1. \end{cases}$$

[1] Note that Fortran also supports `recursive` subroutines, which work in a similar way.

For example, the double factorial of 5 is computed in the following way:

$$5!! = 5 \cdot 3!! = 5 \cdot (3 \cdot 1!!) = 5 \cdot (3 \cdot (1 \cdot (-1)!!)) = 5 \cdot (3 \cdot (1 \cdot 1)) = 15.$$

To implement this function, use a `select case` block. Then, write a `do`-loop that tests your implementation for all **integers** between $n = -2$ and $n = 49$.

- What happens for $n = 20$?
- How would you solve the problem above? Test your approach, but **make sure that your code is compiler-independent**.
- Could you think of another solution that allows for an even larger range? What would be the disadvantage of this approach? *Extra:* Implement and test this approach.

Question 2: Single precision, double precision, quadruple precision, and more?

Link preparation: Scientific Software Development with Fortran - 2.7 (Number Kinds and Precision); Fortran 2003 standard - 13.5.4 (Kind functions), 13.5.6 (Numeric inquiry function) and their specifications in 13.7.

Download `find_real_kinds.f90` from Toledo and inspect the source code. This program searches for the available real kind types using the `SELECTED_REAL_KIND` function. Compile this program using `gfortran`, `ifort` and `nagfor` and run the different executables. Compare the results.
Next, download `print_real_properties_gfortran.f90`, `print_real_properties_ifort.f90` and `print_real_properties_nagfor.f90` from Toledo and inspect the source code. Using the results of `find_real_kinds.f90`, these programs output the following constants for the available real kind types: the kind type parameter, the number of bits in the mantissa, the equivalent decimal precision, the minimal and the maximal exponent, the equivalent decimal range, and the machine precision. Compare the different compilers at your disposal: compile `print_real_properties_gfortran.f90` with `gfortran`, `print_real_properties_ifort.f90` with `ifort` and `print_real_properties_nagfor.f90` with `nagfor`. Answer the following **concluding questions**, discuss your answers with your neighbour(s).

- Which kind types of real numbers are available on the different compilers?
- Does the same kind type have the same kind type parameter on the different compilers?
- What is the machine precision?
- Why would you use kind type parameters and the `SELECTED_REAL_KIND` function?

Question 3: Euler's number

Link preparation: Writing Scientific Software - Section 2, Scientific Software Development with Fortran - 2.7 (Number Kinds and Precision).

It is well known that the sequence

$$\left(1+\frac{1}{1}\right)^1,\ \left(1+\frac{1}{2}\right)^2,\ \left(1+\frac{1}{3}\right)^3,\ \left(1+\frac{1}{4}\right)^4,\ \ldots$$

converges to Euler's number, or in other words

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e.$$

So, if we evaluate $\left(1+\frac{1}{n}\right)^n$ for increasing n, we expect that the result tends to e. We will now verify this numerically using a Fortran program that evaluates $\left(1+\frac{1}{n}\right)^n$ for $n = 10^k$ with $k = 1, 2, \ldots, 20$. For each k, the program should print k, $1+\frac{1}{n}$ and the absolute error $\left|e - \left(1+\frac{1}{n}\right)^n\right|$. Run your program with n in double precision floating point format. Use the `iso_fortran_env` to select the correct kind type parameter. Next, repeat the experiment for $n = 2^k$ with $k = 1, 2, \ldots, 60$.

Answer the following **questions**:

- How do you define the constant e (Euler's number) in the correct precision?
- For $n = 10^k$, does the error converge to zero as expected?
- Why does the error start increasing at $k = 9$?
- What happens for $k \geq 16$? Make the connection with the machine precision.
- What is the difference when using $n = 2^k$?

Question 4: Catastrophic cancellation

Link preparation: Writing Scientific Software - Section 2, Scientific Software Development with Fortran - 2.8 (intrinsic functions)

Complete the following steps and discuss your results with your neighbour(s).

1. Write a Fortran program that evaluates the function

$$f(x) = \frac{1 - \cos(x)}{x^2}$$

for $x = 10^{-1}, 10^{-2}, \ldots, 10^{-16}$ in double precision and that writes the result to the standard output stream. Note that $\lim_{x \to 0} f(x) = 1/2$. Now, answer the following **questions:**

   - What happens for $x \leq 10^{-5}$?
   - What happens for $x \leq 10^{-8}$?

2. Using $\cos(x) = \cos^2(x/2) - \sin^2(x/2)$, it can be verified that f is equivalent to

$$g(x) = \frac{2\sin^2(x/2)}{x^2}.$$

   Compare f and g for $x = 10^{-1}, 10^{-2}, \ldots, 10^{-16}$. **Although f and g are analytically identical, this is not the case when evaluated numerically. Explain why?**

Question 5: Computing the roots of a quadratic polynomial

Link preparation: Writing Scientific Software - Section 2, Fortran 2003 standard - 14 (Exceptions and IEEE arithmetic)

The solutions of the equation $x^2 + bx + c = 0$ can be calculated in two ways:

Algorithm 1	Algorithm 2
$D = \sqrt{\left(\frac{b}{2}\right)^2 - c}$	$D = \sqrt{\left(\frac{b}{2}\right)^2 - c}$
$x_1 = -\frac{b}{2} + D$	$x_1 = \text{sign}(-b)\ \left(\left\lvert\frac{b}{2}\right\rvert + D\right)$
$x_2 = -\frac{b}{2} - D$	$x_2 = \frac{c}{x_1}$

Download the program `quadratic.f90`, that implements both these algorithms in single precision (assuming $\left(\frac{b}{2}\right)^2 - c \geq 0$), from Toledo and have a look at the source code to see how it works. Now complete the following exercises:

- What happens when $b \gg c$, for example $b = 100.0$ and $c = 0.499975$? (The correct results are -0.005 and -99.995) What is the name of the effect you encountered? Can you conclude which algorithm is numerically most stable?

- What happens when $b = c = 0$? How would you solve this?
- *Extra:* Explore the floating point exception handling possibilities from the F2003 standard. Write on the *standard error stream* (its unit number is given by the parameter `error_unit` in the `iso_fortran_env` module) whether the *invalid operation exception* is supported by your CPU. Next, try to detect the zero-by-zero division using the invalid operation flag. For this part of the exercise, you will need to import the modules `ieee_exceptions` and `iso_fortran_env`.
- *Extra:* Check if the rounding mode `IEEE_NEAREST` is in operation. For this part of the exercise, you will need to import the module `ieee_arithmetic`.