dd. May 26, 2018, Arjen Markus

Workshop Modern Fortran, Zürich 6–8 june 2018

Below you will find a number of exercises designed to make use of various aspects of Fortran 2003/2008. The exercises are generally fairly small – after all the emphasis is on using features of modern Fortran, not on learning how to program. Several, however, require more time, because the problem to be solved is more complicated.

As with all programming exercises, these have no "one and only true" answer. You can write a myriad of programs that do not look at all alike that perform the task. We do, however, ask you to use the features of modern Fortran:

- Modules to ensure the compiler can check the interfaces. Also think of the intent for dummy arguments.
- Array operations and properties to clarify the purpose of the code fragment:

  ```
  sumarray = sum( array, array > 0.0)
  ```

  instead of:

  ```
  sumarray = 0.0
  do i = 1,noarray
      if ( array(i) > 0.0 ) then
          sumarray = sumarray + array(i)
      endif
  enddo
  ```

- Derived types and object-oriented techniques to bundle and encapsulate the data instead of spreading them over variables that are related in purpose.
- Memory allocation instead of fixed-size arrays
- Explicit declaration of all variables, to be enforced via `implicit none`.
- Use of the standard routines.
- Use of modern control structures and scoping constructs (`block`, `associate`, internal routines, ...)

Mind you: in these exercises we do not care (much) about performance. Clarity of code is more important.

Recursive functions

Exercise 1: Fibonacci

Write a program to evaluate the Fibonacci sequence from 1 to 1000, that is:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n+2) &= F(n+1) + F(n), \quad n \geq 2 \end{aligned}$$

Use a recursive function for this.

Exercise 2: Q-function

The Q function is defined as:

$$\begin{aligned} Q(1) &= 1 \\ Q(2) &= 1 \\ Q(n) &= Q(n - Q(n-1)) + Q(n - Q(n-2)), \quad n > 2 \end{aligned}$$

It can be found in Douglas Hofstadter's book *Goedel, Escher, Bach*.

Write a program to evaluate this function for n = 1 to 100.

Variant:
It is possible to enhance the performance of this program, by storing the intermediate results.

Working with numerical data

Exercise 3: Determining quartiles

Write a program that can read an arbitrary number of values from a file (the format of the file is simple: one value per line) and then determines the quartiles, that is:

- The value such that 25% of the data read in are below this value.
- The value such that 50% of the data read in are below this value.
- The value such that 75% of the data read in are below this value.

Note: This may require you to sort the data, so you need to store all data before determining the quartiles.

Exercise 4: Handling missing values

Read a data file with numbers, some of which are missing, indicated by "NaN" (not a number). The number of columns is unknown beforehand, but the first column is the independent variable and the lines in the file are at most 100

characters. The program should read the file and produce a table with the independent variable increasing in a fixed step and the other columns determined by linear interpolation.

An example: the input file looks like this:

```
1.0   2.0    2.0
2.0   Nan    3.0
5.0   6.0    Nan
6.0   7.0    7.0
```

Then the expected output (range and step size to be set independently) is:

```
1.0   2.0    2.0
2.0   3.0    3.0
3.0   4.0    4.0
4.0   5.0    5.0
5.0   6.0    6.0
6.0   7.0    7.0
```

Note: This requires the `IEEE_ARITHMETIC` module. As you do not know where the missing values are, you may want to use the `pack()` function per column.

Exercise 5: Finding the closest floating-point number to a root

Floating-point numbers are of course an approximation to the mathematical real numbers. To see how close we can get, consider the following equation:

$$f(x) = e^{-x} - x = 0$$

This equation can be solved for x via, for instance, the Newton–Raphson method:

$$x_{k+1} \quad = \quad x_k - f(x_k)/f'(x_k)$$

The difficulty is to know when to stop. The above equation is well-behaved: at the root the function has a non-vanishing derivative, so with the initial estimate close enough, we will have rapid convergence.

Write a program that determines the root within as much precision as possible with single-precision reals and use the `nearest()` function to examine the function values around the root, that is: make a table of the 20 floating-point numbers surrounding the root you found and the values of the above function.

Exercise 6: Partial differential equations

Consider the following partial differential equation on the domain $0 \leq x \leq 1, 0 \leq y \leq 1, t \geq 0$:

$$\frac{\partial U}{\partial t} \quad = \quad D(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}) + e^U$$

with initial and boundary conditions (D is the diffusion coefficient):

$$\begin{aligned} U &= 0 \quad \text{for } t = 0 \\ U &= 0 \quad \text{for } x = 0, x = 1 \\ U &= 0 \quad \text{for } y = 0, y = 1 \end{aligned}$$

This equation can be solved numerically on a rectangular grid by approximating the partial derivatives via finite differences:

$$\begin{aligned} \frac{\partial U}{\partial t} &\approx \frac{U_{i,j}^{k+1} - U_{i,j}^{k}}{\Delta t} \\ \frac{\partial^2 U}{\partial x^2} &\approx \frac{U_{i-1,j}^{k} + U_{i+1,j}^{k} - 2U_{i,j}^{k}}{\Delta x} \\ \frac{\partial^2 U}{\partial y^2} &\approx \frac{U_{i,j-1}^{k} + U_{i,j+1}^{k} - 2U_{i,j}^{k}}{\Delta y} \end{aligned}$$

Set up a small program that uses array operations to implement the partial differential equation.

Hint: You can use array sections such as `U(2:nx-1,2:ny-1)` to select only a part of the array. This is useful for the boundary conditions and for the finite differences.

Working with collections

Exercise 7: Lucky numbers

A set of numbers that looks asymptotically like primes is obtained in the following way:[1]

- Write down the odd integers: 1, 3, 5, ...
- Then the first odd number larger than 1 is 3. So remove each third number from the above list and obtain: 1, 3, 7, 9, 13, 15, 19, 21, ...
- The first odd number larger than 3 is 7, so strike out every 7th number and obtain: 1, 3, 7, 9, 13, 15, 21, 25, 31, ...
- Continue this process and obtain the list 1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, ...

Write a program to carry out this procedure.

Hint: You can use the function `pack()` and array operations to make a compact program.

[1] `http://mathworld.wolfram.com/LuckyNumber.html`

Exercise 8: Splitting up strings

Given a string of arbitrary length, split it up in pieces. We want a *class* with the following "methods" and properties:

- Set the separator character (only a single one).
- Store a string of arbitrary length, so that we can get the pieces one by one without worrying about the original string.
- Retrieve the pieces in arbitrary order.

For example: the string "A, B,, C " that is split using a comma, consists of the pieces "A", " B", "" and " C " – note the empty substring that is the consequence of the two consecutive commas. Also note the spaces, leading, intermediate and trailing are preserved, as they are not separating characters.

Note: This is best done via allocatable-length strings.

Exercise 9: Linked lists

In languages like C, C++, Java etc. it is very common to use *linked lists* to represent dynamic collections of data. Design a derived type or class that can be used in a similar way:

- Insert data at an arbitrary position in the list – identified by an index.
- Retrieve data from an arbitrary position.
- Delete the data from an arbitrary position.

Hint: Use a recursively defined derived type.

Exercise 10: Vector spaces

One of the nice things about Fortran 90 and later standards is that you can define your own operations. Use this to design an abstract class that can function as a template for *vector spaces*:

- A vector space is a collection of data items, vectors, with two properties:
  - You can add two vectors to obtain a new vector
  - You can multiply a vector with a number to obtain a new vector
- It should be possible to have vectors consisting of scalars, one-dimensional arrays, two-dimensional arrays, single/double-precision reals, complex numbers, etc.[2] This should not matter to the abstract class.
- The class should support the above two operations via user-defined addition and multiplication.

[2] If you are ambitious, you might also consider combining functions in this way.

- It would be nice to have an array of such vectors (all of the same type) and have in addition to the above operations a new user-defined operation, `.in.`, that determines if a vector is already contained in the array or not.

Note: We want an abstract class – not a particular implementation, though this can be used to test the class.

Code modernisation

Exercise 11: Interfaces

BOBYA – minimisation algorithm by Mike Powell
Have a look at the file `bobyqa.f`. It contains the driver routine for the BOBYQA algorithm by Mike Powell for minimising a function of N variables. It was written in a typical FORTRAN 77 style. Study the code and make suggestions for improving the interface:

- Make it possible to better check the arguments (array sizes?)
- Make it more flexible – now the name of the routine that evaluates the function to be minimised is fixed.
- What about checking the correct use of the arguments (intents)?
- Is it possible to reduce the number of arguments?

DGELS – LAPACK routine
The file `dgels.f` contains the driver routine `DGELS` from the LAPACK library. Design an interface routine that makes this easier to use.

Suggestion: Do we really need all the flexibility (arrays `A` and `B`) offered by the original?
PIKAIA – genetic algorithm
Optimisation using genetic algorithms typically requires a lot of options, in this case you can set twelve different parameters. The parameters are stored in an argument `ctrl`. Design a more comprehensive interface, taking advantage of derived types.
ROMIN – minimisation according to Rosenbrock (TOMS 450)
Yet another optimisation routine, this one was written in the 1970s if we can rely on the comments. The code is short enough to allow manual modernisation. Therefore: Use modern constructs and eliminate as many loops and GOTOs as you can.

Exercise 12: A numerical problem

The code in the file `besselk.f` (from the library for evaluating special functions by Zhang and Jin[3]) evaluates the modified Bessel functions of the second kind $K_0(x)$ and $K_1(x)$ [4].

[3] `http://jin.ece.uiuc.edu/specfunc.html`
[4] `http://mathworld.wolfram.com/ModifiedBesselFunctionoftheSecondKind.html`

Unfortunately for $x < 1$, the result is `NaN`. Find out where this is coming from and use the `IEEE_ARITHMETIC` module to improve the interface – add a flag to signal the occurrence of NaNs.

Elaborate exercises

The next two exercises require more extensive programming.

Exercise 13: Queueing systems

Consider the following problem: we need to design a shop (or a help desk or a ticket salespoint) where three employees handle the customers. Customers may take 1, 2 or 5 minutes each and they arrive at random times, but in a continuous stream. The question is whether the customers should pick one of the three counters at arrival or whether they should form a single queue and only when their turn comes up go to the employee who is free.

To make it simpler:

- Option 1: The customer comes and picks a random queue. He/she simply sticks with it, no matter how short the other queues are.
- Option 2: The customer gets in line in the one queue and awaits his/her turn.

The output parameters of this problem are:

- What is the average waiting time for the two options?
- How long are the queues on average with the first option?
- How long is the one queue on average with the second option?
- What is the maximum arrival rate of customers before the queues start growing endlessly? In other words: the capacity of the system.

The assumptions are:

- The customers have an equal probability of requiring 1, 2 or 5 minutes handling time.
- They arrive independently over the day, but there is no pattern. So per hour N customers arrive on average, whatever hour of the day.

Hint: you can use the `events_library` module in the file `events.f90` to facilitate the programming.

Exercise 14: Controlling a stirred vessel

In a factory a product C is produced in a continuous operation from two ingredients, A and B. The reaction is exothermal and the temperature must be maintained between two fairly tight limits, because if the temperature is too low, too little of C is formed and if the temperature is too high, the reaction may run away, causing C to deteriorate.

One problem is that the input stream has a variable concentration of A and B, which can not be controlled. The only control parameter is the heating/cooling of the vessel.

The equations are:

$$\begin{aligned} V\frac{dA}{dt} &= Q_{in}A_0 - Q_{out}A - kAB \\ V\frac{dB}{dt} &= Q_{in}B_0 - Q_{out}B - kAB \\ V\frac{dC}{dt} &= +kAB - Q_{out}C \\ V\frac{dT}{dt} &= Q_{in}T_0 - Q_{out}T + khAB + H \end{aligned}$$

A, B and C are the concentrations of the three substances, T is the temperature of the liquid in the vessel, A_0 and B_0 the concentrations of A and B in the inflow. The coefficient h represents the amount of heat that is produced and H represents the heating/cooling.

As the volume V of the vessel is constant, the inflow Q_{in} and outflow Q_{out} are the same.

The reaction rate k depends on the temperature:

$$k = k_{ref}e^{\alpha(T-T_{ref})}$$

The problem is to simulate this system. The heating and cooling is limited to a fixed value:

$$\begin{aligned} H &= -H_0 \quad \text{if } T > T_{max} \quad \text{(cooling required)} \\ H &= +H_0 \quad \text{if } T < T_{min} \quad \text{(heating required)} \end{aligned}$$

As mentioned, the concentration of A and B in the inflow varies (slowly) over time.

The problem:

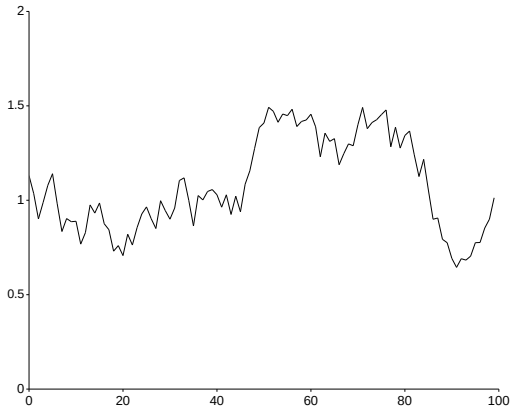
- Create a random function that varies slowly over time (that is: not white noise) around a mean value.

  Use the relation below for example:

$$\begin{aligned} x_0 &= 1 && (1) \\ x_{k+1} &= (1-\alpha)x_k + \alpha x_{mean} + \epsilon_k && (2) \end{aligned}$$

where α is a constant, 0.1, say, x_{mean} is the mean value (or the value around which the series should vary) and ϵ_k is a uniform random value between -0.15 and 0.15 (other values for the parameters are, of course, possible as well).

One realisation of the function looks like this:



- Create a routine that solves the above equations, taking the various parameters as input.
- Use these in a program to actually solve the system and register the total heating and cooling "events". Make sure all the parameters can be read in via a simple input file.

This program can be set up *ad hoc*, only capable of solving the exact problem at hand, but it can be also be organised in such a way that the equation solver is capable of solving aritrary systems of this sort of differential equations. It is up to you, though the second type is preferable.