Introduction to Fortran - Hands On Sessions

Table of Contents

Introductory Comments

- Please consult the web site
  https://doku.lrz.de/display/PUBLIC/Materials+-+Programming+with+Fortran
  for further information on how the exercises should be prepared and conducted. The web site also features downloads for all course materials.
- There is **no need to do *all*** the exercises.
- You are bound to make mistakes. Therefore, writing test programs (comparing expected with actual results) is a vitally important step in the development cycle. And one learns more from the mistakes than from the successes.

Exercises for Day 1

Session 1 (obligatory) – Build and run a simple program

The folder `examples/skel_day1` contains the example program `solve_my_quadratic.f90` (real solutions of a quadratic equation) from the slide talk.

1. Compile and run this program with the various compilers at your disposal. You can use the Makefile that is also available in the `examples/skel_day1` folder ("`make solve_my_quadratic.exe`"), or compile directly via the command line.
2. In order to be able to run a single compiled executable with various values of the coefficients `a`, `b`, `c`, change the program so it repeatedly reads these three real values from standard input. Then, run the program at least with the following combinations:

a	b	c
2.0	-2.0	-1.5
2.0	-2.0	+1.5

   Fix the problem that arises when the second set of values is used. What happens if you supply non-numerical input?
3. Modify the program to deal more gracefully with non-numerical input by catching the I/O error. This can be done by adding an `iostat=<integer_variable>` specification (separated by a comma) in the I/O statement and checking the integer result for being a nonzero value. What other problems might you encounter when reading data from standard input?
4. Replace the "*" format used for output by a format string that displays 10 digits after the decimal point, using scientific notation. Then re-execute the program with

a	b	c
2.0	-2.31	-1.76

The solution for this exercise will be contained in the file `solve_my_quadratic.f90` of the folder `examples/solutions_day1/`.

Session 1 (obligatory) – Sums of squares

Declare an array of integers and write statements to store squares j^2 (j=1, ...) in that array. Then, calculate those integers n for which

$$\sum_{j=1}^{n} j^2 \leq 100$$

What do you need to do if the right hand side of the inequality is 10,000,000,000 instead of 100? How can the required amount of changes to the program be minimized? Print out the resulting list of numbers with a suitably integer-formatted write statement.

The solution for this exercise will be contained in the file `sum_of_squares.f90` of the folder `examples/solutions_day1/`.

Session 1 (obligatory) – Calculating an approximation of π

The program `examples/skel_day1/pi_approx.f90` calculates an approximation of the integral

$$\frac{\pi}{4} = \int_0^{\infty} \frac{dx}{1+x^2}$$

through a discretization process. How is the expression in the summation loop evaluated? Build the program and run it, measuring the execution time via the UNIX `time` command. Modify the program to avoid conversions and both check the results and re-measure the performance.

The solution for this exercise will be contained in the file `pi_approx.f90` of the folder `examples/solutions_day1/`.

Session 1 (obligatory) – Sorting a String

(This exercise is from "Modern Fortran Explained", Section 4.6)

Define a character variable of length 80. Write a program that reads a value for this variable. Assuming that each character in the variable is alphabetic, write code that sorts them into alphabetic order, and prints out the frequency of occurrence of each letter.

Hint: You might want to use the intrinsic function TRIM that removes trailing blanks from a string. For example,

```
character(len=5) :: c
character(len=4) :: a
a = 'w'
c = TRIM(a) // 'x'
```

produces a value of `c` equal to `'wx`*bbb*`'`, while the assignment `c = a // x` would produce `'w`*bbb*`x'` (italicized b indicates a blank).

Session 1 (optional) – Sieve of Eratosthenes

Write a program which calculates all prime numbers between 2 and a given integer, say 100 or 12534 and stores these numbers in an array; once the calculation is complete, print out the results.

Hint: you might want to consult Wikipedia for the algorithm. See
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

The solution for this exercise will be contained in the file `sieve.f90` of the folder `examples/solutions_day1/`.

Session 1 (optional) – Calculate the area of a triangle

This exercise is of interest for people concerned with precise numerics.

Given the lengths of the three sides of a triangle, write a program which calculates the latter's area. A skeleton program is available in the file `examples/skel_day1/triangle.f90`. Try to take care of spurious inputs, for example by skipping the processing of such input data.
Consult http://en.wikipedia.org/wiki/Heron's_formula for information on how to calculate the required quantities.

The solution for this exercise will be contained in the file `triangle.f90` of the folder `examples/solutions_day1/`. If you are interested in the numerical properties of this problem, you might also want to read the paper by W. Kahan, included as `Triangle.pdf`.

Session 2 (part 1 obligatory) – Greatest common divisor

1. The GCD of two integer numbers a, b is the largest integer that divides both numbers without a remainder. To calculate this quantity an algorithm based on that invented by Euclid of Alexandria can be used, which performs the following steps for positive integers a, b:

Step	Action
1	If |a| < |b| swap a and b
2	Perform division with remainder a / b, assign the result to d and the remainder to r
3	If r is zero, go to Step 6
4	Replace a by b and b by r
5	Go to Step 2
6	b now contains the GCD

Write a module containing a module function that calculates the GCD. A main program is provided in `examples/skel_day1/test_gcd.f90`; it invokes the function for a number of datasets and checks the calculated results.

The solution will be in the program files `mod_gcd.f90` and `test_gcd.f90` in the `examples/solutions_day1` folder. The executable can be built via the command `make test_gcd.exe`.

2. (This part is optional) The GCD can alternatively also be represented by the following recursive definition:

   `GCD(a, b) = b` if `a mod b` equals zero,

   `GCD(a, b) = GCD(a mod b, b)` otherwise

   Write a function that uses this definition to calculate the GCD and check against the first implementation for correctness. Which implementation do you expect to be faster?

Session 2 (part 1+2 obligatory) – procedure for solving quadratic equation

Improve on the program from session 1.

1. Write a module procedure as indicated in the talk, and make the main program invoke this procedure; the procedure should return the results in ascending order. Depending on the number of solutions found, zero, one, or two results should be printed. Try compiling an invocation of the procedure with incorrectly typed arguments, or the wrong number of arguments.
2. Add statements to the main program that check the solutions by evaluating the expression $(a \cdot x^2 + b \cdot x + c)/|x|$ via use of an internal function. Run the main program for the following values of the coefficients:

a	b	c
2.0	-2.0	-1.5
2.0	7.4	0.2
0.0	7.4	0.2
2.0	7.4	0.00002

3. (This part is optional) Improve your module procedure by accounting for the degenerate cases. What is causing the problems for the last table row above? How can the relative accuracy be improved? **Hint:** If x_1 and x_2 are the solutions, then the quadratic expression must be equal to $a \cdot (x - x_1) \cdot (x - x_2)$; if one of the solutions is known, the other one can be obtained without requiring a subtraction.

The solution will be in the program files `mod_solver.f90` and `test_quadratic.f90` in the `examples/solutions_day1` folder. The executable can be built via the command `make test_quadratic.exe`.

Session 2 (obligatory) – Calculate a dot product using BLAS

Skeleton code for this exercise is contained in the files `blas77.f90` and `calculate_dot_product.f90` of the folder `examples/skel_day1/`.

The web page http://www.netlib.org/blas/ documents the interfaces for the Basic Linear Algebra Subroutines. Look up the description for calculating a dot product, using default real argument arrays. Then write a program that calculates and prints the **dot product** of the following arrays of size 100:

X: 1.0, 1.0, 1.0, ...

Y: 1.0, 2.0, 1.0, -2.0, 1.0, 2.0, 1.0, -2.0, ...

The program should make use of a manually created explicit interface for the used procedure. An optimized implementation of BLAS is available via Intel's Math Kernel Library (MKL); check your environment for variables whose name starts with "MKL" and make appropriate use of one of these in order to link against the MKL library routines (the provided Makefile makes use of this).

Note 1: There also exists source code for a Fortran 90 style MKL add-on module that contains the interface definitions; if you are interested, look for `blas.f90`, and then identify and compare the interface provided there with your solution. The module uses language features that will be explained later.

The solution will become available as the program files `blas77.f90` and `calculate_dot_product.f90` in the `examples/solutions_day1` folder.

Note 2: a multitude of more complex linear algebra problem solvers is available in the LAPACK library, which is layered on top of BLAS. An implementation of the LAPACK routines is also contained in the MKL.

Session 2 (obligatory) – the PURE attribute

Returning to the GCD code you wrote in session 3 – what happens if you add the PURE attribute to the function you wrote in that exercise and attempt to recompile? If problems appear, please fix them. Otherwise, you can congratulate yourself on your good programming discipline.

Exercises for Day 2

Session 3 (obligatory) – Bodies and charged bodies

Skeleton files are provided as files `mod_body.f90` and `prog_body.f90` of the folder `examples/skel_day2/`.

1. Try building the executable. Why does it fail? What is the simplest way to make it build anyway?
2. Perform the necessary additions of procedures etc. to `mod_body` and `prog_body` to enable the program to build and run **without** removing the `private` attribute from the type definition.
3. Suppose you want to add a further physical property to your objects, namely electrical charge. How might you achieve this without needing to change code that uses the original module, and with only minimal changes to the original module? **Hint:** Apart from necessary expansion of the type declaration in `mod_body`, introduce a second module `mod_charged_body` with suitably configured `use` statements that rename `body` to `charged_body`. Add statements to the test program and convince yourself that both type names can be used simultaneously. Can you assign an object declared with one type name to one declared with the other?

The solutions are provided as files `mod_body.f90`, `mod_charged_body.f90` and `prog_body.f90` of the folder `examples/solutions_day2/`.

Note: The advanced course covers language features (e.g., inheritance via type extension) that make doing the above much more painless and efficient.

Session 3 (obligatory) – Using array intrinsics

(This problem is based on an example in section 8.8 of „Guide to Fortran 2008 programming“)
Suppose a group of 3 students writes 4 tests, and each student receives a score between 0 and 100 (a percentage), as indicated in the following table:

	Test 1	**Test 2**	**Test 3**	**Test 4**
Student 1	85	76	90	60
Student 2	71	45	50	80
Student 3	66	45	21	55

Write a program that stores the scores in a suitably defined array and, using array intrinsics, computes

1. the average score for each test across all students
2. the number of scores above the total average over all scores
3. the average absolute deviation (see http://en.wikipedia.org/wiki/Absolute_deviation for a definition) for each student across all tests. **Hint:** study the compiler documentation for the intrinsic function `spread`.
4. whether any of the students always scored above the overall average.

You may want to temporarily modify the input data to test your statements.
The solution is in the program files `scores.f90` in the `examples/solutions_day2` folder. The executable can be built via the command `make scores.exe`.

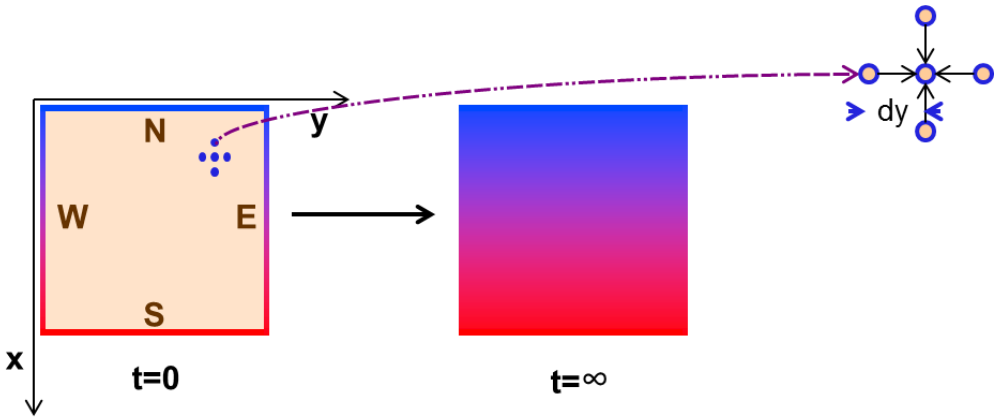
Session 3 (obligatory) – Initializing data

Returning to yesterday's dot product program: Instead of setting up the input vectors in executable statements (with the values specified in the earlier exercise), write data initialization statements that make x and y defined at their declaration. The solution will be made available as the program files `blas77.f90` and `calculate_dot_product_const.f90` in the `examples/solutions_day2` folder.

Session 4 (obligatory) – Simulating heat conduction

Notes: This is a large exercise that is going to be continued later. Skeleton code for this exercise can be found in the files `mod_heat_static.f90` and `heat_static.f90` in the `examples/skel_day2/` folder.

Consider a unit square made of a metal, as indicated by the left hand side of the drawing below:



At the edges of the square, it is clamped to heat sources that keep the temperature constant at 0 on the north edge, at some fixed value Φ > 0 at the south edge, and a linear interpolation between 0 and T on the east and west edges, at all times. At time 0, the inside of the square is assumed to have the temperature Φ=0. The temporal change of the temperature field Φ(x, y, t) is described by the partial differential equation

$$\frac{\partial \Phi}{\partial t} = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2}$$

which over time converges to a stationary solution i.e., one for which the left hand side of the differential equation is equal to zero, as illustrated in the right part of the above figure. The right hand side is also often written as $\nabla^2 \Phi$ (see also below).

1. The temperature field can be simulated by discretizing it along the x and y directions. Establish the necessary declarations in the module `mod_heat_static.f90` for an extended precision temperature field, stored in a private array of rank 2.

2. The boundary (and initial) values for this temperature field need to be set up. Assuming the physicist knowledgeable about the boundary conditions can supply suitable real functions (to be implemented in `heat_static.f90` that calls `heat_bval` and `heat_ival`), complete the implementations of the procedures `heat_bval` and `heat_ival` that take such functions **as arguments** to perform the necessary settings of the temperature field. For the boundary values, an additional argument will be needed that describes which part of the boundary should be defined. The function argument declarations should make use of suitably defined abstract interfaces.
3. The temperature values for time t + δt is obtained from those for time t via incrementation by the temperature differential

$$\delta\Phi = \delta t \cdot \left[\nabla^2\Phi\left(x_i, y_j\right)\right]_{discretized}$$

   Implement that discretization in the module procedure `heat_iter` that performs this calculation for a specified number of iterations; the discretized differential operator can be defined by the stencil approximation

$$\left[\nabla^2\Phi(x_i, y_j)\right]_{discretized} =$$

$$= \frac{\Phi(x_{i+1},y_j)+\Phi(x_{i-1},y_j)-2\,\Phi(x_i,y_j)}{dx^2} + \frac{\Phi(x_i,y_{j+1})+\Phi(x_i,y_{j-1})-2\,\Phi(x_i,y_j)}{dy^2}$$

   Note: the time differential must be chosen such that $\delta t \le 1/(4n^2)$, where n is the number of discretization points along one dimension, for the calculation to converge; this is set in the main program. What needs to be done to assure that the dependencies induced by the stencil calculation are properly accounted for?
4. The main program will need to invoke the iteration procedure repeatedly, until the calculation has converged. Check the correct execution of the code by running a small problem size. Then, run a problem size of 150 x 150 and measure the performance.

The solution for this exercise will be in `examples/solutions_day2/[mod_heat_static.f90, heat_static.f90]`.

Session 4 (obligatory) – Command line processing

The slide talk introduced intrinsic functions for command line processing. In practice, it may be more useful to deploy a convenience library interface, for example one that works like the C libraries' `getopt` interface. An implementation for this is available as a Fortran module in the file `examples/skel_day2/ftn_getopt.f90`. Read the introductory documentation (main page is sufficient) at https://doku.lrz.de/dyn/Doku_Kurse/Fortran/getopt/.

Then, write a program that reads the following command line into appropriately typed variables (making use of the supplied module) and then prints out the result:

```
./example_getopt.exe --len 123 --key 'hello world' --verbose
```

Session 4 (optional) – Prime numbers with arrays

Rewrite the Sieve of Eratosthenes program from day 1 (you can start out from the provided solution if you didn't do the exercise yourself), using array syntax and intrinsics. Note that various solutions are possible. Furthermore, the upper limit should be read from the command line using the previously supplied `ftn_getopt` module, and the necessary array should be established dynamically (**hint:** use an internal procedure). What happens if you attempt to run the program with an upper limit of 100,000,000? If a problem arises, how can one circumvent it?

The solution for this exercise will be in the file
`examples/solutions_day2/sieve_arrays.f90`

Exercises for Day 3

Session 5 (obligatory) - Heat conduction using heap memory

Make changes to copies of the previously written heat conduction source files (or the supplied solutions) that allow you to determine the problem size (i.e., the array dimensions `nx` and `ny`) at **run time**. The following additional steps will be needed to accomplish this:

1. Make the necessary changes to the declarations of the fields `phi` and `phinew`. What else is needed to assure the fields are set up correctly?
2. The x and y directions now may be differently discretized, and the space differentials must also be set up at run time. Use the `ftn_getopt` module to enable command line arguments, e.g.

   ```
   ./heat_dynamic.exe --nx 50 --ny 100
   ```

   for a problem size 50 x 100.
3. Re-do the performance measurements for the problem size 150 x 150.

The solution for this exercise will be in the files
`examples/solutions_day3/[mod_heat_dynamic.f90, heat_dynamic.f90]`.

Session 5 (optional) – Some performance tuning of the heat conduction example

After the stencil calculation, the field `phinew` is copied back to the original field `phi`. The data traffic caused by this copy is unnecessary overhead. How can this overhead be avoided?

Hints: Use POINTER objects. How can you compensate for the optimization reduction caused by use of POINTERs? This version of the program should run faster than the previous one!

The solution for this exercise will be in the files
`examples/solutions_day3/[mod_heat_ptr.f90, heat_ptr.f90]`.

Session 5 (optional) – Transposing a rank 3 array

On day 2, the `transpose` intrinsic was mentioned which allows to transpose rank 2 arrays of arbitrary type. Now, write a procedure which "transposes" a rank 3 array of type real (in the sense that `A(k,j,i) = B(i,j,k)`), avoiding the use of DO loops. The result array should be dynamically allocated inside the procedure.

The solution for this exercise will be in the file
`examples/solutions_day3/transpose_rank3.f90`

Session 5 (obligatory) – Handling rational numbers

Write a module containing an opaque type definition for rational numbers i.e., numbers of the form

$$f = \frac{n}{m}, \text{ with n, m integer numbers.}$$

Implement procedures to set up an object of that type, as well as to deal with various operations which can be performed on such numbers. What pitfalls do you need to avoid for a robust library suitable for production work (Hint: reuse some of the code written earlier in the course)? Make the

facility easy-to-use by introducing suitable overloaded operators. How can you assure that the following code fragment works?

```
type(fraction) :: f
real :: x
integer :: i = 4
f = i
x = f
```

Finally, add a user-defined operator `.inv.` that produces the inverse of a rational number.

As usual, write a test program that checks the correct operation of all your code.

The solution for this exercise consists of the files `rational.f90` and `test_rational.f90` in the `examples/solutions_day3` folder.

Session 5 (optional) – Reduction on fractions

Write an extension of the SUM intrinsic function that works on an array object of `type(fraction)`. It should be capable of operating on arrays of ranks 1, 2, and 3. Only the case of a scalar result needs to be covered.

The solution for this exercise consists of the files `rational_reduction.f90` and `test_rational_reduction.f90` in the `examples/solutions_day3` folder.

Session 6 (obligatory) - Checkpointing a simulation

Add a facility to generate as well as to read an **unformatted** checkpoint file `heat.ckp` for the temperature fields of the heat conduction program. A command line switch should be used to toggle this facility. Reading the checkpoint should of course only be done on the first iteration. Add statements to the module procedure `heat_iter` that handles the writing of checkpoints. Also, modify the module procedure `heat_print` to produce a **formatted** output file `heat.dat` instead of writing to standard output. Data should be written with 13 digits at the end of the computation. Compare the size of that output with that of the checkpoint file.

Hint: the EXIST option of an INQUIRE statement can also be used with a file instead of a unit.

Session 6 (optional) - Heat conduction: control from configuration file

Further improve the usability of the heat conduction example by supporting a configuration file `heat.cfg` that contains NAMELIST entries that control problem sizes, checkpointing options/intervals etc. If a command line argument appears, no use should be made of the configuration file.

The solution for **both** above exercises (checkpointing and input value processing) consists of the files `mod_heat_io.f90` and `heat_io.f90` in the `examples/solutions_day3` folder.

Session 6 (optional) – An internal I/O scenario

Write a subroutine that returns a unit connected to a date-stamped output file, which can be used for logging. The file name should be of the form

`log_<application>_<month>_<day>_<year>.txt`

where „application“ is specified as an argument, and „month“ is a 3-letter string of the form „Jan“, „Feb“ etc. **Hint:** Study the documentation for the intrinsic procedure DATE_AND_TIME.