Fortran Programs from the Ground Up

Note: these examples go from the very simplest to some mildly tricky programs. If you've never used a compiled language, we suggest starting with "Hello World" and going as fast as you like from there. If you're already OK with some Fortran, we recommend skipping to the "Further Suggestions". If you're here for a recap, you can skip directly to the "Real Problems", perhaps pausing at "FizzBuzz" in the "Further Suggestions" on the way.

Guided Fortran Programs

- Hello World, your first Fortran program

  - We're going to do the stock first program, "Hello World" first. Then we'll move on to more useful programs
  - Some people advocate for typing out every line when learning. Do this if you feel it helps you. Otherwise, you can copy-and-paste from the lecture material or your previous programs, but do make sure to read back what you paste and verify it!

  Try:

  Start with an empty text file. Create a PROGRAM main. Include IMPLICIT NONE! We really don't need it without any variables, but we should get into the habit

  Add a PRINT* statement to your program which outputs "Hello World". Compile and run your program. Celebrate!

- Adding some variables

  - Now we can do some simple stuff with numbers

  Try:

  Now start afresh with a new empty text file and create a main program. Remember IMPLICIT NONE! Create an Integer variable called 'i'. Initialise it with a value of (5*4*3*2). Print it out.

  Add a REAL variable to the program. Fill it with a value of SQRT(3). Print it.

- A simple task from scratch

- Now we're going to write a slightly harder code - to calculate the Fibonacci sequence. You can follow the guided steps, or just write the program
- Fibonacci is a "recurrence" relation - describing a sequence where the current term depends on the previous ones. Here s(n) depends on s(n-1) and s(n-2). We start with s(1) = 1 and s(2) = 1 and s(n) = s(n-2) + s(n-1) This gives the "classic" Fibonacci of 1, 1, 2, 3, 5, 8, 13 etc

Guided Steps:

Once each steps compiles and runs, go to the next.

Start with an empty text file and fill in the skeleton of a main program.

Create a loop variable (an integer), and a max_val for the loop. Initialise the latter to 20.

Write a Do loop running from 3 to max_val and print the value of the loop variable.

Create two temporaries for the "previous" pair of values. Call them fib_first and fib_second. They should also be integers. Start them with values of 1. Also create 'fib_current'.

Inside your loop, calculate fib_current = fib_first + fib_second. Now swap fib_first = fib_second and fib_second = fib_current. Output fib_current.

Now you have a simple program!

Add one last step - after you initialise max_val, check that it is valid. It needs to be positive, and you might want to limit the maximum size to e.g. 46. Fibonacci grows FAST - this is the last term

- Churning through some data

- We're going to generate some data from the mathematical Sin function, and do some calculations with it, mainly average, standard deviation (SD) etc.
- The simple formula for SD is

SD = sqrt((1/(n-1)) * SUM{i=1, n} $(a_i$ - <a>$)^2$)

where <a> is the average of the a_i Because it uses the average in each term of the sum in a way that can't be pulled out, this needs two passes over the data

- The alternative "single pass" formula is instead

  SD = sqrt((1/(n-1)) * [SUM{i=1, n} a_i^2]- n<a>^{2})

  Here we can calculate the SUM and the average simultaneously as we pass through our data and at the end generate the SD

Info:

Note that the single-pass option can have problems! If the average is large but the departures from it are small, the accuracy drops. You can demonstrate this by changing from sin to e.g. 1000+sin. On the other hand, it is usually faster as it only needs to access the data once, and does roughly double the calculation-per-access. Modern computers struggle to keep the processor fed with data, so this can be a real benefit for data held in memory (not calculated).

Try:

Calculate sin(2.0 pi x) for x from 0.0 to 2.0 in 1000 bins. Calculate the average and then the Standard deviation (using the 2-pass formula). Do they match your expectation?

Modify your code to also calculate the 1-pass SD and compare the results. Try swapping to 1000 + sin(2.0 pi x). Does the result change? If not, try a bigger number than 1000. To see why this happens, consider what happens if we were to round all the steps to 1 significant figure. Our real numbers are good to about 6 or 7 sig.fig. so don't suffer as badly, but the "sin" part is still much smaller than the "1000"

Further Suggestions

- If you've done all of the above, or already know a bit of Fortran, try some of these programs. Some are really easy, some are a bit tricky. There are "model" solutions for them all on the Github page. These are ONE valid solution - not the only one, but yours should get the same answer at least. Some of the Models demonstrate bits of Fortran you might not otherwise see, so we suggest having a look at a selection of them after writing your own.

- Simple Calculations
  - Calculate the area of a circle given its radius. Take user input using a READ statement
  - Convert Temperatures from F into C or vice-versa. Be careful about Integer division in this one
  - Given a quadratic equation, find its roots. Think about what to do if it doesn't have 2 real roots.
  - Calculate the relativistic gamma function given a velocity. Hint - you might want to consider doing (v/c)^2 instead of v^2/c^2. Make sure to check if v >=c and print an error
- The classic interview problem - FizzBuzz
  - This is a classic "can you program in language x" question, because it uses loops, conditions and simple problem solving. For this reason, it's a really GOOD familiarisation exercise. There are many possible approaches and it's also a fun exercise to see how many you can think up.
  - The brief is: Your code will output the numbers from 1 to 100 in order, BUT if the number divides by 3, it will INSTEAD print the word "Fizz", and if the number divides by 5, it will INSTEAD print the word "Buzz". If the number divides by 3 AND 5, it will print "Fizzbuzz".
- Examples with Big and Small numbers
  - Similar to the Fibonacci program, try writing a factorial(n) program. Note that you can use either an INTEGER or a REAL to store the result and these have different pros and cons. In a plain integer you can get to 12! before the answer is plausible but wrong… In an INTEGER(KIND=INT64) we can go $4x10^9$ further. That's about 8 more terms…! In a REAL, you get a slightly wrong answer, but remain fairly close for a long time. In a REAL64, you can hold very large numbers, but will have deviated a lot from the mathematical value.
  - Calculate sin(x) using the Taylor expansion about 0 (e.g. Wikipedia). This should work up to about pi/4 using the first 10 terms. Be careful with the factorial! Can you break up the terms to reduce the overflowing problems?

- Repeat the previous suggestion but terminate the loop when the answer stops changing by more than 0.1%. This won't converge to the right answer any better, but it will terminate faster.

- Examples with arrays
  - Try writing a Caesar cipher program. Fortran is not great with strings, but it can do most things you'll need. For simplicity, use only upper-case letters. You'll probably want an array of characters. Here's the initialiser to save you typing:

    (/"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", &

    "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"/)

    You can either create a shifted array of the letters first, or you can look up each letter by index and then look up the ciphered letter by looking at index + shift. Remember to wrap around at the ends of the array.

    Remember that to look at or set character 'i' of a string you have to do string(i:i) .

  - Create an array of the values of sin(2.0 pi x) for x from 0.0 to 2.0 in 1000 bins. Now find all of the indices where it changes sign. Check against your prior knowledge of where these should be.

  - Write a program to bubble-sort a fixed length array of data. Bubble sort is on Wikipedia - but basically you take each element and compare it the the next one, and swap if misordered. This means misplaced items "bubble up" (or down) the list.

Real Programs

- These programs assume you know how to program most things, and just want to exercise your Fortran. If you're new to Fortran we do suggest doing one of the Further Suggestions first - in particular try FizzBuzz. Doing something you already know is a good way to get accustomed to the syntax and structure for a new language.

- Numerical Rootfinder

- Code a Newton-Raphson iterative root finder for 3rd order polynomials. Since you'll need both the function and its derivative, you'll want a way to share the parameters between the two. We give 3 possible model solutions for this problem, and there are a host of others. Consider using hard-coded functions for f and f_prime, or using a type to hold a generic polynomial, or using function pointers to "hook up" any functions you wish.
- The Newton-Raphson algorithm says (see e.g. https://en.wikipedia.org/wiki/Newton%27s_method) given an approximate location of a root, xn of a function f, a better approx. is

  $$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

  where f' == df/dx
- Given an initial guess, you can repeat this iteration a fixed number of times, or stop when the answer stops changing more than some threshold. NOTE: Newton-Raphson does not always converge. The model solutions demonstrate a situation where it doesn't.

- The Game of Life
  - This is a classic example of where simple rules lead to very complex behaviour. The core of the code is a N x N array of cells which may be "dead" or "alive".
  - At each iteration, the state of cells is updated according to the states of its 8 immediate neighbours (left-right, up-down and diagonally) so that:
    1. Any live cell with fewer than two live neighbours dies
    2. Any live cell with two or three live neighbours is unchanged
    3. Any live cell with more than three live neighbours dies
    4. Any dead cell with exactly three live neighbours becomes a live cell
  - You can start with random states, or set up one of the simplest moving structures, the glider. See e.g. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life . The glider is (0 is dead and # is alive):

    0 # 0
    0 0 #
    # # #

- There is a simple Ascii-art display module in the Github materials to output a pretty result, as well as some other helper stuff, described at the end of this Document.

- Ising Spin model
  - This is a simple model for “magnetic domains” in solid materials. Individual atoms in the material have a “spin”, like a tiny magnet, free to point in either direction, and for neigbouring atoms these can be aligned or counter aligned. Alignment reinforced the tiny fields to create a bigger one, anti-alignment makes them cancel. Since the material is not at absolute-zero, the spins “jiggle” and there is a chance they flip direction. This gets more likely the hotter they are.
  - We suggest using an integer array of size N x N, containing values of +1 or -1. Start with all 1 or with alternating (checkerboard), or with random states. All give interesting answers.
  - We iterate to get the system to “relax” to an equillibrium (this is NOT time evolution, we’re just trying to find a stable state)
  - We iterate for T trials. For each trial:
    1. Pick a random cell (a random x, and an *independent* random y position)
    2. Calculate the Energy change if this cell were to be flipped. This is given by:
       1. delta_E = J * spin(cell)* Sum_over_neighbours{ spin(neighbour)} with J a parameter
       2. For a single spin-flip, this is equivalent to calculating the energy before and the energy after and subtracting them
    3. If delta_E < 0, flip the spin always
    4. If delta_E > 0, flip the spin with a probability of exp(-beta * delta_E) where beta is a parameter
    5. Leave a strip of cells round the edge of the domain untouched OR use periodic boundaries, where the left wraps around to the right etc

- The parameters are:
  - beta: inverse temperature (1/T). High beta is cold, low beta is hot
  - J : Usually given a value of ±1, this dictates whether spins want to align or anti-align. For J = 1, spins align, and this makes a ferromagnet. For J=-1 the spins alternate. This is called an anti-ferromagnet.
- Step 3 happens when it is energetically favourable for the spin to be in the other direction, and this **always** happens
- Step 4 depends on beta, i.e. the temperature. If beta is very small, the probablity approaches 1. This is a "hot" material where the spins jiggle a lot. If beta is large, the probability is lower, and for "infinite" beta it approaches 0. This is a "cold" material, where there is little or no thermal jiggling.
- There is a simple Ascii-art display module in the Github materials to output a pretty result, as well as some other helper stuff, described at the end.

- A Linked List model code
  - This sort of approach comes up in a lot of physics and chemistry problems for dealing with particles in a physical domain. It's also useful for keeping track of subsets of items which need different handling, where copies are contraindicated (speed, size etc)
  - The linked list is a sequence of nodes, each of which has links to it's previous and next nodes. The list can be "walked through" by starting at the first node and jumping to each in turn. But, you cannot go to a particular node by number (no random access).
  - Write a simple doubly-linked list structure, and functions to add at it's end, and to remove a given item from it. You'll probably want a type for each node and a type for the list. The links between nodes must be pointers. See the ExamplePointer.f90 program in the models for a walkthrough of basic fortran pointers. s
  - Use your linked list in a dummy physics problem. I chose to have a list of particles, and shuffle them between lists depending on whether they were 'fast' or 'slow'. This is convenient, because I can work with either list, or link them head to tail and work with the complete list either.

Appendix - Helper modules for the Real-World codes

For some of the codes, you might want to use some trickier features, so we have some example code to smooth this over. These examples are also handy demos of some of the more obscure or advanced bits of Fortran.

kinds.f90 - supplies the KIND parameters for basic numeric types (integers, reals). The names match those used in the F2008 standard, so using these lets you smoothly swap over to this standard if you wish. F2008 is almost well supported enough for this, but some systems may not support it.

random_mod.f90 - Fortran provides a basic random-number function, RANDOM_NUMBER. This module wraps this function and deals with seeding the RNG before starting. You can simply call "random()" and get a new random number between 0 and 1

sleep_mod.f90 - C provides mili-second resolution sleep functions, but Fortran does not. This module provides two functions, sleep_sec(seconds) to sleep for a given number of seconds, and millisleep(milliseconds) to sleep for a given number of milliseconds. The code itself also demonstrates a simple use of C interoperability

ascii_display.f90 - provides functions to display 2-D logical and Int16 arrays in Ascii-art, with borders, titles etc. This isn't too hard, but is fiddly because it uses non-advancing (no line breaks) IO. This also provides a very simple "wait for any key" function

command_line.f90 - From Fortran 2003, you can access command line arguments easily, but we find it a bit fiddly to do the parsing and the type conversions. This module wraps all this away. You just ask for an argument by name, and supply a variable into which the value is put. The argument is parsed as the type you ask for. See the command_line_snippet.f90 code for an example of use. This code also demonstrates how to create an overloaded interface that dispatches to a suitable function by type.