THE UNIVERSITY OF BRITISH COLUMBIA

Physics 210 Assignment # 7: FORTRAN

Tue. 26 Oct. 2010 — finish by Tue. 02 Nov.

You now know how to "program" in various "programming environments" — `bash`, `muview`, `gnuplot`, `extrema`, `matlab` and `octave` — and in a full-featured "scripted language", `python`. There are many, *many* more of both, and by now you should sense that a given task can probably be implemented in any one of them, as long as it is "Turing-complete",[1] but each has its own strengths and weaknesses.

One weakness of scripted languages is that the computer's CPU has to *interpret* the instructions before it can execute them. This takes time, so if you want your algorithm to run faster you should get the interpretation out of the way ahead of time and make a *binary executable* version of your program in *machine code* that your CPU can execute directly. This is called *compiling*, and all applications that involve massive, time-consuming calculations are written in compilable languages. The most common of these are `C` and `C++`, but the oldest and in many respects the simplest is `FORTRAN`, which is still the most widely used for certain types of computational physics. So this week we will introduce ourselves to `FORTRAN`.

On `hyper`, the `FORTRAN` compiler is `gfortran`, which stands for the `GNU` implementation of `Fortran 95`. (There is a lot of idiosyncratic history here — big surprise....)

As usual, create your `/home2/phys210/$USER/a07/` directory and put all your finished results in it.

1. **TELL A `fib`:** Recall the **Fibonacci numbers**? The file `~phys210/fib.f` contains the `FORTRAN` source code to generate a sequence of Fibonacci numbers, starting from one specified by the user and continuing for an additional number of Fibonacci numbers also specified by the user.[2]

   (*a*) Go through the file `fib.f` and **add comments** that explain what each step is doing. Be verbose! Remember, in `FORTRAN` any line that starts with a `C` or a `c` is a *comment*, and is ignored by the compiler.

   (*b*) **Compile** your edited source code with all the comments using the command

   ```
   gfortran fib.f -o fib
   ```

   which creates an executable image in the file `fib` (if you left out the "`-o fib`" part, it would put the executable image in the default file `a.out`).

   (*c*) **Try it out** using the command "`./fib`" If it works, you are done with this part. If not, go back to step (*a*). `(`I may have made a mistake. `:-)`

2. **LINFIT PROGRAM:** Using what you have learned about `FORTRAN` from `fib.f` (plus lots of additional *Google*-ing and reading, no doubt), "**port**" the `python` code you wrote last week into a `FORTRAN` program `linfit.f` (source code) and compile the `linfit` executable image. Your program should ask a question like, "Where is the file with the data you want read in?" It should read and interpret your answer, read in the data from the specified file, calculate the best linear fit, print out the results, and exit.

3. **LINFIT SUBROUTINE:** You may notice that there are two types of things going on here: a bunch of input/output ("I/O") and a calculation that could easily be generalized and used in a lot of different situations. When this happens, it is tempting to put the generalizable part into a **subroutine** which is "called" from the main program when the data is all read in and ready to fit. To do this we separate our source code into two files, the first (`linfit_main.f`) containing the main program with its I/O specifics and a line like

   ```
   call linfit (npts, x, y, w, p0, p1)
   ```

   and the second (`linfit_sub.f`) containing the subroutine "source" code, starting with the lines

   ```
   subroutine linfit (npts, x, y, w, p0, p1)
   integer npts
   real x(*), y(*), w(*), p0, p1
   ```

   ... and ending with...

   ```
   return
   end
   ```

   Generating the executable image is only slightly more complicated now that it has to be assembled from two components. See if you can guess how to do this. (Try the most obvious guess first!)

4. **GO TO THE LIBRARY:** When we get a *lot* of subroutines and functions[3] it becomes useful to *store* the components as *object modules* (`*.o`) in a *library* like `libmy.a` (a "static" library) or `/usr/lib/libmyutils.so` (a "dynamic" or "shared" library). Here you will only need to manage the first

[1] *Google* it!

[2] Also recall the *factorials* script in Assignment 3.

[3] A **function** is like a subroutine that returns just one "answer" — compare `linfit` which returns two: `p0` and `p1`. If you declare a function like `function squared (x)` you might call it in the form `y = squared(x)` in your main program.

type, which are used like a grocery store: you select the ingredients you need from the inventory, take them home and make a tasty meal from them. The latter type are more like pizza delivery: when you want to run a program that uses modules from a shared library, the required modules are installed at run time and can be (as the name suggests) *shared* by many other tasks running simultaneously. If you want to learn more about the latter, *Google* `shared dynamic libraries`.

To learn about *static* libraries, just say "`man ar`" — because the program that manages static libraries is called `ar` (for "archiver"). All you really need to know are these three steps:

(*a*) Generate an object module `linfit_sub.o` from the source code `linfit_sub.f` using the `gfortran` compiler:

```
gfortran -c linfit_sub.f
```

(*b*) Insert (or replace) the `linfit_sub.o` in the `libmy.a` library:

```
ar rcs libmy.a linfit_sub.o
```

(*c*) Build your program (again with `gfortran`, retrieving `linfit_sub.o` from `libmy.a`:

```
gfortran linfit_main.f libmy.a -o linfit
```

[4]

Do it. Save all your results in the usual place, along with all your source code, object modules and executable files.

5. `make` **IT SO:** All this coding and compiling and archiving and building and installing can get very complicated, especially when you are building an application that uses object modules from many libraries, and *particularly* when you are making changes to required subroutines and functions in different libraries and trying to keep the assorted libraries (and the final product) all up to date! Although it would be wonderful if one "spoke Geek" well enough to give all these instructions to the computer in the right order every time without any goofs, any reasonable person would want to have (at least as a "backup") some sort of stored script that remembers it all for you — a *program* in its own right, in other words, but this time a program for *building an application*.

In Linux this script is called a `Makefile`, because the program that interprets the commands in the `Makefile` is called `make` and, as usual, it is invoked by the command "`make`". See (*e.g.*) `http://en.wikipedia.org/wiki/Make_(software)` or *Google* "Makefile" for details.

The syntax of a `Makefile` is extremely arcane. Some consider it to be the ultimate in "Geek lingo". Nevertheless it is possible to adapt a working `Makefile` to `make` your own project, by just changing the names of the programs, libraries and sources. If you copy the files `fib_main.f`, `fib_sub.f` and `Makefile` from `~phys210/` to your own `/home2/phys210/$USER/a07/` directory, `cd` to that directory and enter the terse command

```
make
```

you should get a new version of the `fib` executable and a new file called `libmy.a` containing the `fib_sub.o` object module. Then, without necessarily understanding exactly what the `Makefile` is doing,[5] **edit and adapt** your copy of the `Makefile` to add `linfit_sub.o` to the `libmy.a` static library and, from it and `linfit_main.f`, compile and build your `linfit` executable.

Once this works, you can make any modifications you like to either `linfit_main.f` or `linfit_sub.f` and generate a new, up-to-date, executable `linfit` just by entering "`make`".

[4] Note how clever `gfortran` is: you don't need to tell it *which* object module(s) you need from the `libmy.a` library (which may archive many!) — it will go look for whatever it is missing in that archive.

[5] Very few users of `Makefile`s have any idea how they work when they first try one out! But after a few adaptations to their own purposes, they begin to get a fair idea — and this is enough for the time being.