Beginner Fortran 90 tutorial (part 3)

1 "Canned routines"

Very often, you will find yourself in a situation where you want to use a routine or a function created by another person. To do so, you will have yo to find this routine on the web, or in a book, and understand it well-enough to embed it in your own program.

A good example source for such routines is the Numerical Recipes library. You can either copy the routines from the book itself, or from an online website. The key to using such canned routines is to understand:

- Exactly what they do
- How to call them, what parameters do they need, what items do they return?
- What are the dependencies of the routines (i.e. are they standalone, or do they need other routines or module to function)
- How stable/reliable they are (i.e. under which circumstances are they expected to work, and when are they expected not to work)

In a *good* routine source, you should find all of this information in the documentation provided. If not, you'll have to go digging into the program to understand what it does. Finally, once you understand how the routine works, you should test it on a case for which you know the answer before applying it to problems for which you do not know the answer.

In this section, we will learn to use and test the `gaussj.f90` routine of Numerical Recipes to find the inverse of a matrix $\mathbf{A}$.

Exercise 1: Either copy the routine from the book, or find it online. Read the instructions very carefully and try to answer the following questions:

1. What does this routine do?
2. What are the input and output parameters required in the calling sequence. How are these related to the linear algebra problem you are interested in solving?

3. What subroutines/functions does it call, what modules does it use?

The answers to all of these questions are not all trivial. Carefully read the book and related material on `gaussj.f90`.

Exercise 2: Download or copy all of the additional modules and subroutines you need, and put them in the same directory as `gaussj.f90`. Modify the program from the last homework to call `gaussj.f90` instead of your own Gauss-Jordan elimination routine. Make sure you give the program a different name (you will need to keep your old program as a point of comparison). Create a `Makefile` to compile your new program, `gaussj.f90` as well as all of the required dependencies. Try to compile it. Correct whatever errors come up, and repeat as necessary until you have a program that does compile.

Exercise 3: Run your old program and the new program using the same input matrix files for **A** (note that you *will* have to give `gaussj.f90` a **B** matrix – just give it any vector of your choice). In each case, print the inverse of **A** to the screen (note that you need to make sure you understand where and how the solution is stored). Do you obtain the same answer? If so, hurray (you're done). If not, could the error be due to partial vs. full pivoting (i.e. does it look like a small truncation error or is it more fundamental)? If the latter, debug and repeat as necessary!

This illustrates that it can sometimes take longer to understand how someone else's code works than to write one yourself. However, well-documented and well-tested publicly available routines can be very useful!

2 Routines available in a pre-compiled library

In the example given earlier, the entire routine `gaussj.f90` was available for you to read, modify if desired, etc.. and the way in which it is included in the program is very similar to the way in which you would include one of your own routines in the program.

Routines coming from commercial libraries on the other hand rarely work like that. Usually, these libraries are pre-compiled for your particular computer. You just have to call the routine from your own program, and *link* the library to the program using a compiler command. The main differences with what we did earlier is that (1) you sometimes don't even get to see the code for the routine you're using, and (2) you don't link your program to the actual routine, and the routine is not in the same directory as the main code. Instead, you link your program to a *library* and that library is usually held centrally somewhere on the computer you are using. This way, you do not have to copy the entire library into your working directory every time you want to create a new code that uses it.

In what follows, we will learn to search for, use, and link routines from the LAPACK (and BLAS) libraries. Both are very commonly-used, highly-optimized routines for linear algebra problems.

Exercise 4: Search the LAPACK library for a routine that will return the inverse of a matrix. Be careful find one that works for any real matrix (for your own education, look at what other options are available for more specific kinds of matrices). What do you notice?

Exercise 5: It is quite clear from the documentation that this routine is not a standalone routine : instead, it takes the output from another routine to calculate the inverse of **A**. Algorithmically, this means that we will have to have our program call `sgetrf` first, and then `sgetri`. Read the documentation, and try to answer questions 1-3 of the previous section for both of the routines.

Exercise 6: Modify the program used earlier to use the LAPACK routines instead. This time, you can ignore the matrix **B**. As before, have the program print the resulting inverse matrix to the screen.

Once the program is written, comes the tricky part of linking it to the libraries. Take a deep breath, because this can unfortunately be a very frustrating process the first time around. Typically, the compiler command will look something like:

`gfortran` (*the usual list of all the code files you need*) (*a command that tells the compiler where the library is*) `-llapack -lrefblas`

Suppose that your code only uses your normal `LinAl.f90` module, as well as a program called `testLAPACKinverse.f90`. On the `grape` cluster, the libraries are located in the directory `/scratch/lapack-3.5.0/`. The correct compile command would then be (in one line)

```
 gfortran LinAl.f90 testLAPACKinverse.f90 -L/scratch/lapack-3.5.0/
-llapack -lrefblas -o name.exe
```

To put it in a Makefile instead, you could add:

```
testLAPACK:
        gfortran LinAl.f90 testLAPACKinverse.f90 -L/scratch/lapack-3.5.0/
-llapack -lrefblas -o mytest.exe
```

making sure the space in front of the commands are tabs, not separate spaces. You can then compile the code with the simple command `make testLAPACK`.

Now this will work fine if you are on `grape`. If you are on another machine, you will *first* have to find out where the `LAPACK` and `BLAS` libraries are (note that they are usually stored together, because the `BLAS` library is needed to compile the `LAPACK` one). You may even have to install them yourself. To do so, don't panic just yet, go to `www.netlib.org/lapack/`, and try to follow the instructions. I'm afraid I have no idea how to install these libraries on a Windows system (so if that's your only option you are now allowed to panic) but it is pretty straightforward to do it on a Unix or Mac system. Just make sure you write down *where* you install them (anywhere is fine, I think), because the `lapack-3.5.0` directory the installation creates is the one you will have to refer to in the compiler commands above. Note that you have to install the `BLAS` libraries first, and then the `LAPACK` ones. The installation, if successful, should create 2 files in the directory `lapack-3.5.0`: the file `librefblas.a` and the file `liblapack.a`. These two files are the two compiled libraries you need!

Exercise 7: Modify your `Makefile` as described above to compile the new program. If it compiles, hurray, otherwise, come and see me. Once it does eventually compile, compare the output of your new code with the one from the two other codes your created .... Is the answer what your hoped for?

You are now an expert in using canned routines and commercial libraries! You are ready to start the homework!