APC 524 / AST 506 / MAE 506

Software engineering for scientific computing

Clarifying notes about the API in Assignment 1

What's in this document

Task A of Assignment 1 asks you to refactor one big `main()` function into several separate functions that have different responsibilities, but it asks you to still keep all those functions in a single source file.

Would it be better to move those separate functions into separate source files? Generally speaking, the answer is 'yes'. Even if you will only ever have one version of an ODE integrator routine, keeping that in a separate file from, say, your printing routines substantially facilitates the housekeeping associated with code organization and code maintenance. The principal reason why you aren't asked to do a multifile refactoring in Assignment 1 is to keep the assignment manageable. But in principle, once you've done a good job on Task A, your code should be sufficiently modular that splitting it into several source files wouldn't be too daunting.

In Task B, you're asked to move *some* of the code – the part responsible for executing the integration of the ODEs – into a separate source file. In that case, it's not for overarching housekeeping benefits but rather to facilitate making that sub-code conform to an interface so that it can be swapped out without needing to rewrite any other parts of the code (an interface is sometimes called an "API", or Application Programming Interface).

For those for whom programming to an interface is new, and because I rushed through this last week, I will try to clarify the point below, first by elaborating on the housekeeping benefits of multi-file project layouts in general, and then describing how using multiple files furnishes us with one way (but by no means the only way) of implementing an API.

Some benefits of multiple source files

Suppose I had only asked you to move the integration sub-code into a separate source file for housekeeping purposes. Then you would:

- make a new source file `integrator.c` with the integration code;
- place the declarations for entities defined in `integrator.c` into a header file `integrator.h`;
- `#include` that header file in `ddo.c` so that those entities can be referenced and used within `ddo.c` without the compiler throwing a fit;
- create separately compiled object files `ddo.o` and `integrator.o` that get *linked* together into an executable.

The *clerical* benefit of modularity across several files is that, if we need to make implementation changes later in `integrator.c`, we don't have to touch `ddo.c` or recompile it. We'd only recompile `integrator.c` and then link the updated `integrator.o` with our pre-compiled `ddo.o` to make a new executable. Particularly when source files become millions of lines long, you don't want one small change in one source file to obligate you to recompile every file in the project.

What if I have different variants of a source file?

But now suppose I want to write *numerous* different implementations of the integration code (e.g. `integrator_01.c`, `integrator_02.c`, ...), and that sometimes I want to build an executable that uses one implementation, but other times I want my executable using a different implementation. [1] If all these source files contained functions with slightly different signatures, or used totally different looking data structures, I'd probably have to maintain correspondingly different versions of my driver program (e.g. `driver_01.c`, `driver_02.c`, ...), each of which would `#include` a different header (`integrator_01.h`, `...02.h`, `...03.h`, ...), and our bookkeeping would get rapidly unmanageable.

API to the rescue

But perhaps with some forethought, we could come up with a common signature for the pertinent functions across our different integration files, and a common "containering" for the data types they'd each need to do their work, and so forth.

For instance, perhaps function `do_stuff_01(int n, double x, double y)` in `integrator_01.c` performs essentially the same "big picture" task as the (differently signatured) function `do_stuff_02(int n, double *x, double *y)` in

[1] We're imagining a scenario where I'd always be using only one of those implementations in a given executable.

`integrator_02.c`, just with a different implementation. If it's no skin off our nose to rewrite the internals of `do_stuff_01()` so that it shares the same signature as `do_stuff_02()` (i.e. takes pointers as input instead of actual doubles), and if we'd only ever be using one or the other of those two functions in an executable but not both, then we could:

- name these functions `do_stuff(int n, double *x, double *y)` in *both* source files;
- put that single signature into a single header file `integrator.h`;
- likewise come up with a common "face" for everything else in our integrator source files (other functions, containers for data structures, etc), and put those corresponding declarations into `integrator.h`; and
- have both `integrator_01.c` and `integrator_02.c` use that common header file as their bridge to the outside world.

A common header file of this sort is just one way of implementing the more abstract notion of an API. It lets you have multiple variants of one "code chunk" without a bookkeeping explosion. We only need *one* header file `integrator.h` and one driver file `driver.c` that `#includes` that header and gets compiled into an object file `driver.o` *once*. We then build a whole *library* of compiled object files (`integrator_01.o`, `integrator_02.o`, `...`) and build different executables by linking `driver.o` to whichever version of the integrator we want.

This is what's happening in Task B, only instead of calling our different implementation files `integrator_01.c`, `integrator_02.c`, `...`, we're giving them the more descriptive names `euler.c`, `rk4.c`, and `ab2.c`. And `integrator.h` is the header file common to all of them.

Separate code *providing* a service from code *using* that service

More important than the bookkeeping, the API makes it substantially easier to write code that *uses* an integrator as part of some larger task. The API establishes a contract between the code that *provides* the service of integrating ODEs from any other code, even within the same project, that *uses* that service (a.k.a. client code). It gives the author of the client code the luxury of conceptualizing tasks in high-level terms ("...then I get my parameters, then I integrate the resulting ODEs, then I capture the output, then...") without being burdened by (or even needing awareness of) the low-level details of how each of those tasks gets accomplished (which is part of the problem with "pipeline-y" code such as that found in `nasty.c`).

The practical value of this kind of separation is a central lesson of this course.