APC 524 / AST 506 / MAE 506

Software engineering for scientific computing

Assignment 1: Code refactoring and APIs

Assigned: 28 Sep 2019

Due: 11 Oct 2019, 11:55pm

This initial assignment (which must be programmed in C) has two major components: Tasks A & B (detailed below). Realistically, once you get underway with Task B, it might make you rethink and adjust your solution to Task A. That's certainly ok, and even encouraged, but not required (while hindsight from the latter will certainly inform the former, the two tasks are independent).

Detailed background about the damped, driven oscillator and about ODEs is provided in an Appendix at the end of this document.

A made-up backstory to set the mood

Consider a damped driven oscillator system

$$\ddot{q} + 2\beta\dot{q} + \omega_0^2 q = \cos(\omega t) \quad , \tag{1}$$

with $\omega_0 \equiv 1$ and $\beta \equiv 0.25$ (treat these parameters as fixed). The oscillator will be driven by a unit-amplitude forcing function whose driving frequency ω can be adjusted between runs.

You are working on a research project that requires you to integrate this system numerically for a fixed total time $T = 6\pi$, always starting from rest at equilibrium. During the course of your work, you'll need to vary two things: the integration timestep[1] Δt (or, alternately, since the total integration time is fixed, the number Nsteps of timesteps taken from $t = 0$ to $t = T$), and the driving frequency ω.

You have inherited an executable (prepared by a previous student whom no one knows how to contact anymore) that generates the integration output for a few different combinations of ω and Nsteps. That executable is generated from a single C source file `nasty.c` that has just a single "pipeline-y" function `main()`. While not *wrong* (in the sense of its outputs), this "code" is a hot mess. It leaves much to be desired in terms of structure, style, and design (too "pipeline-y", lots of copy-pasted repetitive code, no modularity, no comments, etc).

Over the next several weeks, we will be making incremental (and eventually sweeping) changes to this code in order to explore some basic principles of software design as well as to get some practice with coding in low-level languages like C and C++.

[1]Assume all time steps during a single integration are equal sized, but that the size of that timestep may need to change between runs.

Task A: Make `nasty.c` less of a hot mess.

I'm being somewhat open-ended about what this means, in part because this whole exercise is somewhat artificial in the first place, but here are some requirements/constraints to help you narrow the scope:

- Keep everything in one source file for now, called `lessnasty.c` (but definitely refactor the code into separate functions that have clearly demarcated responsibilities);
- Compile `lessnasty.c` into an executable `ddo` that accepts ω and Nsteps as command-line arguments (in that order);
- For each (ω, Nsteps) pair for which the current nasty code outputs results, those results should come out the same when the same ω and Nsteps and are fed into the new executable;
- Currently, `nasty.c` integrates everything using the simple first-order forward Euler method. Your new version should likewise use only forward Euler;
- **Your executable must be able to build from source on Adroit, using `gcc` with C11 compatibility.** Just because you are able to build it on your laptop, don't assume it will build on Adroit without explicity confirming this for yourself (Mac users take special note here).
- Don't go overboard with error-checking. Error checking is nice, but there are more pressing lessons that this assignment is here to teach. So check that the user is inputting the correct number of command-line arguments, but other than that, don't really worry about it for Task A.

To clarify – you're *not* expected to achieve some Platonic ideal of refactored code. After all, having all your functions in one file is already suboptimal, as is the lack of a well thought-out global structure for this mini-project that facilitates subsequent maintenance and extension. Rather, Task A is here in case you aren't used to thinking about redundancy, extensibility, legibility, etc and/or if you have limited experience refactoring nasty code that you didn't originally write.

Besides the refactoring, you should also try to spot and rectify other glaring no-nos in `nasty.c`, add some explanatory comments so there's at least a modicum of documentation, and so forth.

You'll be graded on three counts:

(i) your code must compile (on Adroit, using gcc);

(ii) for each (ω, Nsteps) pair, your executable must give the same output as the nasty code;

(iii) your refactoring and design choices will be assessed (fairly holistically, sometimes with attention to specific choices that you do/don't make).

For Task A, the AIs will assess (iii) even if (i) and/or (ii) fails.

Remember, once the assignment is returned and we go over common pitfalls or bad choices, **you'll be able to resubmit** (within a narrow time-window) **to get up to half the missing points back.** So think of this as a feedback mechanism to help you calibrate. Many of you already think about structure and design and don't need this exercise (in which case, it should be fairly quick). But some of you do, and this is the fastest way for us both to find that out.

Task B: Write an ODE-integration library that adheres to an API.

More fake backstory

You've successfully refactored the crappy file you inherited so that it's more manageable. But now your advisor informs you that, going forward, she wants to be able to change the integration algorithm without having to recompile the part of the code that handles everything else (the processing of command line arguments, encoding the ODEs for the oscillator, printing output, etc).

The natural solution is to specify a common *interface* for code that implements numerical ODE integration. To gain clarity on why, stop reading this, take 5 minutes to read the accompanying short note `APInotes.pdf`, and then come back to this document. I'll wait. . .

Welcome back.

The API

So we need an API. In this assignment, you won't be asked to design it – your advisor already got some postdoc to do it, in a header file `integrator.h` that all source files holding integration code will share:

```c
#ifndef INTEGRATOR_H
#define INTEGRATOR_H

/*
 * Pointer to a function that computes the time derivative of x
 * Dx = f(x,t) ,
 * where n is the dimension of the solution x.
 *
 * Should return 0 if successful, nonzero if error.
 */
typedef int (*FuncPtr)(int n, double t, const double *x, double *Dx);

typedef struct integrator_t Integrator;

/*
 * Return a new Integrator object
 *
 * n:   dimension of state vector
 * dt:  timestep
 * rhs: pointer to a function to compute right-hand side of
 *      Dx = f(x,t).
 */
Integrator *integrator_new(int n, double dt, FuncPtr rhs);

/* Free the Integrator object and any associated memory */
void integrator_free(Integrator *integrator);

/*
 * Advance x by one timestep
 *
 * Should return 0 if successful, nonzero if error.
*/
int integrator_step(Integrator *integrator, double t, double *x);


#endif /* INTEGRATOR_H */
```

You ***cannot alter this file!***

To-do list for Task B

- Make a copy of your source file `lessnasty.c` from Task A and name it `ddo.c` (for Damped Driven Oscillator);
- Move the integration code from `ddo.c` into a separate source file `euler.c` and adjust the entities in that file as necessary so that the file conforms to this API;

- `#include` the API header file into `ddo.c` (so that all entities it declares can legally be invoked within `ddo.c`, despite being defined in a different source file);
- After Task A, `ddo.c` presumably contains a function that encodes the ODEs for our oscillator system. You'll need to adjust the signature of this function so that it also conforms to the API;
- Modify the driver code in `ddo.c` so that, rather than call an integration *function* as you presumably had it do after Task A, it creates and uses an `Integrator` as specified in the API;
- Make sure that both `ddo.c` and `euler.c` compile, that you can successfully link `ddo.o` and `euler.o` (and other things, like the standard math library) to make an executable `ddo_euler`, and that this executable produces the same output for a given (ω, Nsteps) pair as `ddo` did before Task B even began;
- Write additional source files `rk4.c` and `ab2.c` that implement, respectively, 4th-order Runge-Kutta and 2nd-order Adams-Bashforth integration schemes (the pertinent math background is found in the Appendix of this assignment), all conforming to the API[2];
- Make sure *those* files compile successfully (but don't recompile `ddo.c`! If it worked correctly before, you shouldn't need to.);
- Build two more executables `ddo_rk4` and `ddo_ab2` by linking `ddo.o` to `rk4.o` and `ab2.o`, respectively;
- You should probably check that these new executables are giving sensible answers by, e.g. , integrating our ODE system with routines in Scipy or MATLAB. Not required, but a good sanity check, since if your code doesn't give reasonably correct outputs, grading stops there.

You'll be graded on four counts:

(i) your code must build (on Adroit, using gcc);

(ii) for each (ω, Nsteps) pair, your `ddo_euler` executable must give the same output as the nasty code did in Task A;

(iii) all your executables must give reasonably accurate and properly outputted answers for a few (ω, Nsteps) pairs that we won't tell you in advance;

[2]For clerical ease and modularity, you may end up choosing to define "helper functions" in `rk4.c` and/or `ab2.c` that are only used by other functions within those source files. That's ok. But each of `rk4.c` and `ab2.c` *must* define all the entities required by the API, and any helper functions you end up writing in these files should never be used outside those files.

(iv) ***if* you pass the steps above,** then the code itself will be assessed (again, fairly holistically, sometimes with attention to specific issues).

Makefiles

Even with up-arrow, compiling and recompiling your code on the command-line every time you change something or have to debug is annoying, not to mention error-prone. Moreover, it's easier than you think to forget that, after changing file `X`, you now also have to recompile file `Y`, because it depended on `X`. Modularity helps us have less of this drudgery but cannot fully eliminate it.

`make` is a utility that helps automate the build process. To simplify your life, the starter code for Task B includes a file called `Makefile` that has pattern-based instructions for `make` about how and when to compile things for you (NB: the Makefile doesn't apply to and won't work for Task A. For Task A, you'll need to do all compiling yourself on the command line, but Task A has just a single source file, so not too annoying).

We haven't yet discussed how `make` works, but you can still utilize it for Task B as follows:

- To, say, compile `euler.c` into `euler.o`, just type

  ```
  $ make euler.o
  ```

  on the command line (and analogously for `rk4.o`, `ddo.o`, etc).

- To build the executable, say, `ddo_rk4`, type

  ```
  $ make ddo_rk4
  ```

  If `rk4.o` (or anything else on which `ddo_rk4` depends) needs to be re-built first, the rebuild will happen automatically (that's the magic of `make`). Any linking to the standard math library will be handled automatically. Proceed analogously for the other executables.

- Only the files you're supposed to be building for Task B are programmed into the `Makefile`. So, unfortunately, typing

  ```
  $ make me-a-sandwich
  ```

  will just give an error.

- To build every executable required by Task B (and any object files needed en route to those executables), type

  ```
  $ make all
  ```

- To remove all the `.o` files and executables (i.e. to clear your working directory of everything except source code and headers), type

  ```
  $ make clean
  ```

Helpful hints

- **REMEMBER:** there are no `euler.h`, `rk4.h`, or `ab2.h` header files – all these modules use `integrator.h` as a single common header file. If that statement is surprising, then you didn't go read `APInotes.pdf` when I asked you to a few pages ago. Bad!
- While I won't explicitly show you how to modify `ddo.c` (since this is part of what you need to figure out), I *have* included a working version of `euler.c` to help guide you. In other words, *part of Task B is done for you*.
- I recommend you get to the point where you have a working code that can build a correctly functioning executable `ddo_euler` from `ddo.c`, `integrator.h`, and `euler.c`. Once you have that, move on to the other integration algorithms.
- The details of what exactly an `Integrator` contains may (or may not) vary among `euler.c`, `rk4.c`, and `ab2.c`, and likewise for the definitions of other API entities across the different source files. The rationale behind this approach is that `ddo.c` doesn't know or care about such implementation details (because it shouldn't need to). More to the point, once you have a working `ddo_euler` executable, you shouldn't need to recompile the `ddo.o` object file ever. All you'd need to do is generate a new executable by *linking* `ddo.o` with one of `rk4.o` or `ab2.o`. The runtime outputs of the resulting executables would presumably differ both from each other and from what `ddo_euler` produces (since you're using a different integration algorithm), but the previously compiled `ddo.o` object file ought to continue to work as-is.
- **Mind your memory** – part of this assignment is to think about hardware considerations, in particular stack vs. heap memory. Think about where you want the memory for different constructs to come from. Sometimes there's no right answer, but you should be making informed and deliberate choices.

  For instance, even though this assignment doesn't make use of the following feature, your implementations should allow concurrent use of multiple different integrators of a given type (e.g. a different driver program should be able to spawn, say, 7 distinct instances of an RK4 integrator to integrate 7 different ODE systems in parallel).

Submission

Details about how to submit the assignment will follow soon (Dev set up the system, and he explained it to me, but I don't quite remember the details, so one of us will explain it shortly).

But once those logistics are resolved, your submission should include the following (and only the following) files:

Task A

- `lessnasty.c`

That's it. Remember, we should be able to build this into an executable on Adroit, and we will. Do ***NOT*** submit your executable yourself.

Task B

- `Makefile`: (the same file we gave you) We will build your executables using this file.
- `integrator.h`: (the same file we gave you) the provided file specifying the interface.
- `ddo.c`: your driver program to integrate the damped driven oscillator (and also handle command line arguments, print output, etc)
- `euler.c`: implementation of the Euler integrator (same file as what was provided to you)
- `rk4.c`: implementation of the 4th-order Runge-Kutta integrator.
- `ab2.c`: implementation of the 2nd-order Adams-Bashforth integrator.
- `README.md` (optional this time): It's generally good practice to include a plaintext markdown file briefly describing your software, how to build it, & how to run it.

Appendix

Math background: systems of ODEs

A common task in scientific computing is to integrate a system of first-order ordinary differential equations (ODEs)

$$\begin{aligned} \dot{x}(t) = g(x(t), t) \quad , \\ x \equiv (x_0, x_1, \ldots, x_{n-1})^T \in \mathbb{R}^n \quad , \\ g \equiv (g_0(x,t), g_1(x,t), \ldots, g_{n-1}(x,t))^T \in \mathbb{R}^n \end{aligned} \tag{2}$$

with an initial condition $x(0) = (x_0(0), x_1(0), \ldots, x_{n-1}(0))^T$. We focus on systems of first-order ODEs because pretty much any set of ODEs of any order can be brought into this form through an appropriate change of variables.

Example: the simple harmonic oscillator

The simple harmonic oscillator

$$\ddot{q} + \omega_0^2 q = 0, \qquad q \in \mathbb{R} \tag{3}$$

is a 2nd order ODE in the single variable q. But after the change of variable $p \equiv \dot{q} \to \dot{p} = \ddot{q} = -\omega_0^2 q$, it becomes a coupled first-order system

$$\begin{aligned} \dot{q} &= p \\ \dot{p} &= -\omega_0^2 q \end{aligned} \tag{4}$$

that matches the form (2) with $n = 2$:[3]

$$\begin{aligned} x(t) = (q(t), p(t))^T \,, \qquad x_0 \equiv q, x_1 \equiv p \\ g(x,t) \equiv (p, -\omega_0^2 q)^T = (x_1, -\omega_0^2 x_0)^T \quad . \end{aligned}$$

Example: the damped driven oscillator

A (one-dimensional) damped driven linear oscillator has the general form[4]

$$\ddot{q} + 2\beta\dot{q} + \omega_0^2 q = f(t) \quad , \tag{5}$$

where β is the damping constant (same units as ω_0), ω_0 is the natural frequency of the *undamped undriven* oscillator, and $f(t)$ is the driving force (per unit

[3] ω_0^2 is just a fixed parameter, not a variable.

[4] I am using the notation found in *Classical Mechanics* by John Taylor (specifically equation (5.48) of that textbook).

mass). Via an analagous change of variable, this 2nd order ODE can likewise be written as a coupled first-order system

$$\begin{aligned} \dot{x}_0 &= x_1 \quad , \\ \dot{x}_1 &= f(t) - \omega_0^2 x_0 - 2\beta x_1 \\ \text{i.e. } g_0(x,t) &= x_1 \quad , \\ g_1(x,t) &= f(t) - \omega_0^2 x_0 - 2\beta x_1 \quad . \end{aligned} \tag{6}$$

Background on integrator algorithms

There are many methods for numerically approximating the solution to a system of first-order ODEs. In what follows, a subscript j on an x denotes the whole x vector, evaluated at time t_j.

Forward Euler method The simplest/crudest is the *forward Euler* (a.k.a *explicit Euler* method, a first-order-accurate method given by

$$x_{j+1} = x_j + hf(x_j, t), \tag{7}$$

where h is the timestep and x_j denotes the value of the (n-dimensional vector) solution at time $t = t_0 + jh$ (usually we take $t_0 \equiv 0$).

Runge-Kutta methods (4th order) Perhaps the most widely used integration scheme is a *fourth-order Runge-Kutta* method, given by

$$x_{j+1} = x_j + \frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4, \tag{8}$$

where

$$\begin{aligned} k_1 &= hf\left(x_j, t_j\right) \\ k_2 &= hf\left(x_j + \frac{k_1}{2}, t_j + \frac{h}{2}\right) \\ k_3 &= hf\left(x_j + \frac{k_2}{2}, t_j + \frac{h}{2}\right) \\ k_4 &= hf\left(x_j + k_3, t_j + h\right). \end{aligned}$$

In essence, RK4 boils down to taking four distinct Euler steps of size h, but each starting from a different (x,t) condition. The results of these four Euler steps are stored as the k_i values (note that computing k_{i+1} requires us to first compute k_i). Each of these four distinct steps has different associated higher-order error terms, but the particular linear combination of the k_i chosen in (8) causes all the error terms up to fourth order to cancel.

Adams-Bashforth methods (2nd order) Another widely used scheme is a second-order *Adams-Bashforth* method, given by

$$x_{j+1} = x_j + \frac{3}{2}hf(x_j, t_j) - \frac{1}{2}hf(x_{j-1}, t_{j-1}) \tag{9}$$

Note that, unlike the previous two schemes, this is a *multi-step* method, which requires the solution values x_j and x_{j-1} at the previous *two* timesteps in order to compute the solution x_{j+1} at the next timestep. As a result, the solution x_1 at the end of the *first* timestep must be handled by some other method (since there aren't two time steps worth of output on which to rely yet). Typically, x_1 is calculated using an even higher-order scheme (such as 4th-order Runge-Kutta), but **for this assignment, you should calculate the output at the end of the first timestep with the explicit Euler method given above.**

Regardless of which scheme is used, note that, according to the API, an `Integrator` object shouldn't *do* anything; rather, it should hold state information (including possibly pointers to functions) that an `integrator_step()` function can use (along with the current values of t and x) to compute the solution value x at the *next* timestep.