APC 524 / AST 506 / MAE 506

Software engineering for scientific computing

Assignment 2: Intro to object-oriented programming

Assigned: 23 Oct 2019
Due: 03 Nov 2019, 11:55am

Summary

This assignment expands on the task from Assignment #1 — integrating an initial-value problem (system of ODEs) forward in time — but this time using C++, which allows for a fully object-oriented approach that is less "kludge-y" than the C version.

An object-oriented approach will streamline how we represent (and subsequently adhere to) interfaces. In Assignment #1, the API in `integrator.h` did double duty, specifying both how `Integrator`s should work *and* (indirectly, via the `FuncPtr typedef`) what the signature of `rhs()` functions should be. In contrast, this assignment will separate out the API for the ODE systems into its own abstract base class called `Model`. Doing so will permit us to swap in/out not just different `Integrator`s but also different `Model`s, so that we can integrate any system of ODEs with any algorithm.

Moreover, the internal state of `Integrator` and `Model` objects will let us handle parameters better (without having to do unpleasant things like define global variables). Finally, to be more general, this time we will build a single executable that lets us specify both our ODE system (including parameters and initial conditions) and our integration scheme at runtime.

Provided files

Your starter repository contains **six files:**

- `integrator.h`: defines an abstract base class `Integrator` for ODE integrators. **This class will never be directly instantiated.**
- `euler.h`: defines a concrete derived class (concrete subclass) of `Integrator` called `Euler` that encodes an explicit Euler integrator.
- `euler.cc`: The implementation file associated with `euler.h`. Remember, in C++, it's good practice to put info pertinent to *using* a class in a header file (the public members and declarations of the public methods) and segregate the *implementation* of methods to an eponymous `.cc` file.

- `model.h`: defines an abstract base class `Model` for representing $\mathbf{f}(\mathbf{x}, t)$, the right-hand side of a generic system of ODEs (see the Appendix of Assignment #1). **This class will never be directly instantiated.**
- `ddo.h`: defines a concrete subclass of `Model` called `DDOscillator` that encodes the damped-driven oscillator from Assignment #1.
- `ddo.cc`: The implementation file associated with `ddo.h`.

Assignment

The assignment has **five elements:**

1. **Version control** — you must track your work on the assignment using git. Even though you will pick a final commit to grade, your overall repo history will also be assessed. View this as a low stakes opportunity to practice using git on a whole small project. Fancy git acrobatics are not required; you don't even need to branch. Yes, there are probably natural points at which you'll want to make branches and then either merge back into or rebase onto `master`, but it's up to you. Do make sure your log messages are coherent and informative, and try to make your commits on any given branch fairly atomic (but not microscopically atomic).
2. **Write new Integrators** — using the `Euler` class as guidance, and based on your experience with Assignment #1, write classes `RungeKutta4` and `AdamsBashforth2` (both concrete subclasses of `Integrator`) to implement 4th order Runge-Kutta and 2nd order Adams-Bashforth integrators (the underlying math is exactly as in Assignment #1).

   DIFFERENCE FROM ASSIGNMENT #1: this time, you should take your first Adams-Bashforth step using an RK4 step rather than an Euler step.

   As with `Euler` instances, instances of these classes should have their timestep and their `Model` fixed once and for all upon initialization. Changing either of these things should require creating a new object.

   This code should live in files `rk4.h`, `rk4.cc`, `ab2.h`, and `ab2.cc`, as appropriate.
3. **Write a new Model** — using the `DDOscillator` class as guidance, write a new class `LotkaVolterra` (a concrete subclass of `Model`) to implement the right-hand sides of the Lotka-Volterra *nonlinear* ODEs (these equations describe predator-prey dynamics – see the Appendix to this document for more information). Like `DDOscillator`, the class `LotkaVolterra` should allow arbitrary parameter values, with different values of the parameters corresponding to different instances of these classes.

4. **Driver code** — using your `ddo.c` file from Assignment #1 as a logical springboard, write a general driver `ode_solve.cc` in C++ that lets you choose between the different equations and integration schemes at runtime. **For this assignment, continue to keep all code other than the class definitions in the single source file** `ode_solve.cc`**.**

   Beyond classes (and associated things like using `new` and `delete` rather than `malloc` and `free`), you don't need to make explicit use of any other features of C++ (though you are welcome to, if you know, e.g., what `std::vector` means). You don't even need to use `std::cout` — you can continue to use regular old `printf()` and everything should work fine. In short, the driver code for this assignment can be mostly regular C . . . except with classes.

   DIFFERENCE FROM ASSIGNMENT #1: since the ODE system can now change, fixing the total integration time or any parameters of the system in advance makes little sense, so your command-line arguments will now specify a stepsize + a total number of integration steps, along with a suite of system parameters and initial conditions for the integration.

   More specifically, you should generate a single executable `ode_solve`, and it should accept command-line arguments in the following format:

   ```
   $ ./ode_solve <model> <params> <ICs> <integrator> <timestep> <numsteps>
   ```

   where:

   - `<model>` is one of the strings "ddo" or "lv"
   - `<params>` is a space-separated list of parameters for `<model>`, enclosed as a standalone quoted string (see the example command-line below for clarification). These parameters should be in the same order in which they will be passed to a `Model` constructor. If there are no parameters for `<model>`, this command-line argument should be the empty string "".
   - `<ICs>` is a space-separated list of initial conditions consisting of

     $$t,\, x_{\text{init}}[0],\, x_{\text{init}}[1],\, \ldots,\, x_{\text{init}}[n-1]$$

     enclosed as a standalone quoted string (see the example command-line below for clarification)
   - `<integrator>` is one of the strings "euler", "rk4", or "ab2"
   - `<timestep>` is the timestep (floating point)
   - `<numsteps>` is the number of integration steps to take (integer).

So, for instance, to integrate the damped-driven oscillator[1] as in one of the runs from Assignment #1, with $\omega = 2.0, F = 1.5, \beta = 0.25, \omega_0 = 1.0$, starting at rest from $x = 1.5$ at $t = 3.0$, using an Euler integrator with a timestep of 0.1 for 40 steps, you would enter (see the constructor signature in `ddo.h` for the order of the oscillator parameters)

```
$ ./ode_solve ddo "2.0 1.5 0.25 1.0" "3.0 1.5 0.0" euler 0.1 40
```

Don't worry about error-trapping command-line arguments too heavily.

Output: Integration output should follow the same format as in Assignment #1: at each time, print t, followed by all the state variables, all space separated, each formatted as `%15.8f`, all on one line, one line per time point.

5. **Makefile:** supply a makefile named `Makefile` that has targets to:

   - generate object files for each of the classes listed above;
   - generate object files each of the extra classes that you have to write;
   - build the executable `ode_solve` (you should be able to specify either `ode_solve` as a target or `all` as a target to generate the executable);
   - purge your directory of object files and executables (a `clean` target).

   Your makefile does *not* have to be pattern-driven the way the makefile you were given for Assignment #1 was. But it can be – feel free to copy liberally from that other makefile, but make sure you undersand what you're copying or you'll commit errors.

MAKEFILE DIFFERENCES FROM ASSIGNMENT #1:

- You're only making one executable, and *all* the object files should be linked to generate that executable. So if you decide to write a pattern-driven makefile, you shouldn't just straight copy the makefile from Assignment #1.
- You'll compile using the *C++* compiler `g++`, rather than the *C* compiler `gcc`. Therefore, the pertinent compiler flags that `make` uses should be stored in a variable called `CXXFLAGS` rather than `CFLAGS`.
- To specify the C++17 standard, you'll need to add a flag `-std=c++17` (everything you need to do should actually work fine with C++14 or even C++11, but for those of you who know C++ and want to use more of its features, let's establish common ground and compile with the C++17 standard)

[1] We've added a parameter F to the damped-driven oscillator. It denotes the amplitude of the driving force. In Assignment #1, this was always fixed to unity. Now, we'll allow it to vary.

- Ditch the `-pedantic` compiler flag — we don't really need it, and it might cause some issues with C++.
- If you use `patsubst`, remember that our source files now have the `.cc` suffix, not `.c` (object files still have the `.o` suffix).

Grading

You'll be graded on four counts:

(i) your git repo should follow general good practices for solo work. That means you can't just work elsewhere and end with a single commit+push at the end that has your final work. Though cramming things in during the last two days is probably a bad idea, we won't really be looking at your timestamps. It's more about having a reasonable DAG, meaning log messages, branching and merging as you feel is appropriate, etc, all assessed fairly holistically. It's an opportunity to practice good solo version control.

(ii) your code must build (on Adroit, using `g++`) using your makefile. **NOTE!: The default version of `gcc/g++` on Adroit is from 2015. To make sure you are using a more recent version of `g++` that understands C++17, first do `$ module load rh/devtoolset/8` as explained here.**

(iii) when the `Model` is the damped-driven oscillator(s) whose parameter values, initial conditions, timestep, and number of steps correspond to a scenario from Assignment #1, your executable should give the same outputs (to several decimal places) when using integrator X as when using integrator X under that scenario in Assignment #1;

(iv) we will check the accuracy of your Lotka-Volterra integrations against some known solutions. You can validate your code as usual by integrating the system in Python or MATLAB to validate results, but here's one example as as gut check: with ICs –> (t=0, prey=1.0, predators=2.0) parameters ($\alpha = 1.0, \beta = 0.2, \delta = 0.1, \gamma = 0.2$), and a total integration time of 50, a reasonably accurate integration like RK4 with 200 timesteps should show that the predator and prey populations are both periodic (max prey population ~ 8.5, max predator population ~ 10.5) with around 3 cycles each over that timespan, and the oscillations a bit out of phase (predator peaks lag, i.e. come after, prey peaks). Plotting results to validate your integrations will help.

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

Helpful hints

- We're not making separate executables this time – any source file can `#include` any of the headers, so any of your extra classes can be "aware" of any of the others. If you think about what the substeps in the Runge-Kutta or Adams-Bashforth procedures actually entail, leveraging other classes could help streamline code. You're not required to have your classes use other classes, but it's an interesting way to think about the ODE integrators in this assignment.
- If you parsed command-line arguments directly in `main()` in Assignment #1, I wouldn't do that here. It's going to get a bit involved, so better to outsource that to a separate code block within `ode_solver.cc`.
- There are several ways to sub-parse the ODE parameters and the initial conditions, with varying degrees of sophistication. For this assignment, it doesn't much matter what you do to parse the command-line arguments, as long as it works.

  For those who will be mostly writing C-esque C++ (C++ can use pretty much all C library functions as-is), the function `strtok()` in `<string.h>` lets you parse a string into substrings. Check out this link or google other examples. You have to do a little looping, but if you use a counter, you can also count how many substrings you end up with, and stick all the parameters into an array.

  Admittedly, much of this is sub-ideal. For instance, I've structured the constructor for `DDOscillator` so that all the parameters have to be `doubles`, but you could imagine a scenario where some parameters are of other types. Don't worry about having your driver code be able to adapt to lots of different ways of listing parameters for a `Model`. Just assume any parameter list for a `Model` will be an array of `doubles`. This is just a homework assignment, and we're sacrificing some generalizability re: parameters for the sake of getting you to build something with a panoply of concrete classes that can be interchanged at run time (an example of the "Strategy" design pattern).

Submission

You will clone your personalized repo for this assignment from github (via AG350), and you will submit by pushing back up to github and identifying one commit as your final submission (but your whole repo history will also be evaluated, since git-ing is part of this assignment).

Your submission commit should include the following files:

- `integrator.h`

- `euler.h`, `euler.cc`
- `model.h`
- `ddo.h`, `ddo.cc`

(all of the above should be the same files that appear in the starter version of the repo, unless there turns out to be some error and I send out corrected versions)

- `rk4.h`, `rk4.cc`
- `ab2.h`, `ab2.cc`
- `lv.h`, `lv.cc`
- `ode_solve.cc`
- `Makefile`
- `README.md`: Overwrite the default `README.md` found in the starter repo. Your `README.md` should be a Github-flavored markdown file with a brief summary of what the classes do and how they're used, the build instructions for your executable (i.e. "Type `make blah`"), and the basic syntax/usage of the driver program `ode_solve`. This doesn't have to be long, but it should capture the essentials. You do *not* have to document or explain either the integration algorithms themselves or any implementation details – the README is a quickstart guide for *users* of the code.

Appendix

The Lotka-Volterra equations

The Lotka-Volterra equations define a simple model to describe the evolution of two interacting populations over time: a population of predators (say, foxes) and a population of prey (say, rabbits). The premise is that predators consume prey and convert consumed prey into new predators (via reproduction) with some efficiency. Meanwhile:

- rabbits are assumed to be otherwise immortal
  - they have unlimited food sources;
  - they mainly die when eaten by foxes (because their natural lifespans are orders of magnitude longer than the mean time until a fatal encounter with a fox, even when the fox population is small);
- foxes only eat rabbits and starve to death if the rabbit population becomes too small.

To model this quantitatively, let $F(t)$ and $R(t)$ denote the sizes of the fox and rabbit populations, respectively, at time t in some convenient unit like thousands of individuals (e.g. $F = 2$ denotes a population of $2 \times 1000 = 2000$ foxes). Moreover, define the following parameters:

- α: the *per capita* birth rate of rabbits (number of rabbits born per unit time per each existing rabbits);
- β: the *per capita* death rate of rabbits due to predation by a single fox (number of rabbits eaten per unit time per fox per each existing rabbit);
- δ: the *per capita* reproduction rate of foxes for each rabbit consumed (number of foxes born per unit time per fox per each rabbit consumed). This is the efficiency with which the fox species converts hunted rabbits into new foxes.
- γ: the *per capita* death rate of unfed foxes (number of foxes who die per unit time per each existing fox, absent food).

Then the aforementioned dynamics can be captured by the following system of ODEs:

$$\begin{aligned} \dot{R} &= \alpha R - \beta RF \\ \dot{F} &= \delta RF - \gamma F \quad . \end{aligned} \tag{1}$$

This *nonlinear* system of coupled ODEs has two equilibrium points:

- $(R = 0, F = 0)$ – if both populations are zero, they pretty clearly remain zero
- $(R = \frac{\gamma}{\delta}, F = \frac{\alpha}{\beta})$ – for these values of R and F, the derivatives $\dot{R}$ and $\dot{F}$ both vanish simultaneously.

In particular, the latter equilibrium turns out to be an elliptic fixed point, so that initial conditions in a (pretty large) neighborhood of the fixed point in the (R, F) state space yield oscillatory solutions for both $R(t)$ and $F(t)$.