APC 524 / AST 506 / MAE 506

Software engineering for scientific computing

Assignment 2: Object-oriented integrator

Assigned: 7 Oct 2014

Due: 14 Oct 2014, 11:55pm

This assignment will approach the same task as that of Assignment 1, integrating an initial-value problem forward in time, but using an object-oriented approach in C++. This will allow us to easily change the system to solve, and the integration scheme to use. We will then conduct a convergence test, comparing our numerical solutions to those of an exact solution.

Differential equations

As before, we will solve the system of n ordinary differential equations, of the form

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t), \qquad \mathbf{x} \in \mathbb{R}^n. \tag{1}$$

We will consider three different particular systems. The first is the forced *Duffing oscillator* from Assignment 1:

$$\ddot{x} + \delta\dot{x} - x + x^3 = \gamma \cos \omega t \tag{2}$$

for $\delta = 0.2$, $\gamma = 0.3$, $\omega = 1$, for initial conditions $x(0) = \dot{x}(0) = 0$. The second is the famous *Lorenz system*::

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy, \end{aligned} \tag{3}$$

with $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$. The third is a forced linear oscillator of the form

$$\ddot{x} + 2\beta\dot{x} + x = \gamma \cos \omega t \tag{4}$$

for $\beta = 0.1$, $\gamma = 1$, and $\omega = 0.9$.

Integration schemes

We will use the same three integration schemes used in Assignment 1: explicit Euler, 4th-order Runge-Kutta, and 2nd-order Adams-Bashforth. Recall that the *explicit Euler* scheme is given by

$$\mathbf{x}_{j+1} = \mathbf{x}_j + h\mathbf{f}(\mathbf{x}_j, t), \tag{5}$$

where h is the timestep, and $\mathbf{x}_j = \mathbf{x}(jh)$. The fourth-order *Runge-Kutta* scheme is given by

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \frac{h}{6}\mathbf{k}_1 + \frac{h}{3}(\mathbf{k}_2 + \mathbf{k}_3) + \frac{h}{6}\mathbf{k}_4, \tag{6}$$

where

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_j, t_j)\\
\mathbf{k}_2 &= \mathbf{f}(\mathbf{x}_j + (h/2)\mathbf{k}_1, t_j + h/2)\\
\mathbf{k}_3 &= \mathbf{f}(\mathbf{x}_j + (h/2)\mathbf{k}_2, t_j + h/2)\\
\mathbf{k}_4 &= \mathbf{f}(\mathbf{x}_j + h\mathbf{k}_3, t_j + h).
\end{aligned}$$

The second-order *Adams-Bashforth* is a multi-step scheme with

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \frac{3}{2}h\mathbf{f}(\mathbf{x}_j, t_j) - \frac{1}{2}h\mathbf{f}(\mathbf{x}_{j-1}, t_{j-1}). \tag{7}$$

Recall that this scheme is not “self-starting,” as it needs both $\mathbf{x}_0$ and $\mathbf{x}_1$ in order to compute $\mathbf{x}_2$. In the first assignment, you used the first-order-accurate explicit Euler method for the first timestep (i.e., to calculate $\mathbf{x}_1$). This is actually a poor choice, as the future error is often dominated by the error at the first timestep. *Here, you should use fourth-order Runge-Kutta to compute the first timestep for Adams-Bashforth.*

Exact solution

In order to validate our schemes, we will compare our numerical solutions to an exact solution, for the forced linear oscillator above. (The other two systems are actually *non-integrable*, and there is no exact solution.) The exact solution of (4) is given by

$$\begin{aligned}
x(t) &= e^{-\beta t}(c_1 \cos\omega_d t + c_2 \sin\omega_d t) + A\cos\omega t + B\sin\omega t\\
\dot{x}(t) &= e^{-\beta t}\big[-\beta(c_1\cos\omega_d t + c_2\sin\omega_d t)\\
&\qquad\qquad + \omega_d(-c_1\sin\omega_d t + c_2\cos\omega_d t)\big]\\
&\quad - \omega(A\sin\omega t - B\cos\omega t),
\end{aligned}$$

where $\omega_d = \sqrt{1-\beta^2}$ is the damped natural frequency, the coefficients A and B are given by

$$A = \frac{(1-\omega^2)\gamma}{(1-\omega^2)^2 + 4\beta^2\omega^2}, \qquad B = \frac{2\beta\omega\gamma}{(1-\omega^2)^2 + 4\beta^2\omega^2},$$

and the constants c_1, c_2 are determined by the initial condition:

$$c_1 = x(0) - A, \qquad c_2 = \big(\dot{x}(0) + \beta(x(0) - A) - \omega B\big)/\omega_d.$$

To test convergence, we will use the two-norm of the error. If $\mathbf{x}_j$ denotes our numerical solution at timestep j, and $\hat{\mathbf{x}}_j$ denotes the exact solution, then the two-norm of the error is

$$\|\mathbf{x} - \hat{\mathbf{x}}\| = \left(\sum_{j=0}^{N} |\mathbf{x}_j - \hat{\mathbf{x}}_j|^2 h \right)^{1/2} \tag{8}$$

where h is the timestep, N is the number of steps taken, and $|\mathbf{x}_j|$ denotes the (Euclidean) norm of the vector $\mathbf{x}_j$,

Assignment

We will provide a code to integrate the Duffing oscillator using an explicit Euler scheme. The provided files are the following:

- `model.h`: an abstract base class `Model` for evaluating $\mathbf{f}(\mathbf{x}, t)$, the right-hand side of (1).
- `integrator.h`: an abstract base class `Integrator` for ODE integrators
- `duffing.{h,cc}`: an implementation of the Duffing oscillator (a derived class of `Model`).
- `euler.{h,cc}`: an implementation of an explicit Euler integrator (a derived class of `Integrator`).
- `duffing_solve.cc`: a driver program to integrate the Duffing oscillator for the given parameter values and initial conditions.
- `Makefile`: builds the program `duffing_solve`.

The assignment is divided into five parts:

1. Write classes `RungeKutta4` and `AdamsBashforth`, both derived classes of `Integrator`, to implement Runge-Kutta and Adams-Bashforth integrators. Note that the Adams-Bashforth integrator should use Runge-Kutta as the first step. Deliverables: `runge-kutta.{h,cc}`, `adams-bashforth.{h,cc}`.
2. Write classes `Lorenz` and `LinearOscillator`, both derived classes of `Model`, to evaluate the right-hand side of the Lorenz equations (3), and the linear oscillator (4), for arbitrary parameter values. You should also modify your main routine as needed to test this, and take the initial conditions to be $(0, 0.01, 0)$. Deliverables: `lorenz.{h.cc}`, `linear-oscillator.{h,cc}`.

3. Write a general driver program `ode_solve` that lets you choose between the different equations and integration schemes. It should read command-line arguments in the following format:

   ```
   ode_solve <equation> <integrator> <timestep> <numsteps>
   ```

   where `<equation>` is one of `{duffing, lorenz, linear}` and `<integrator>` is one of `{euler, rk4, ab2}`. The parameter values should be those specified above, and the initial conditions should be zero for Duffing and the linear oscillator, but $(0, 0.01, 0)$ for the Lorenz system. The output should be as in the provided `duffing.cc`, with time in the first column, and an additional column for each state variable. Deliverable: `ode_solve.cc`.

4. Write a program `test_convergence` to do a convergence test of the different integrators. The parameters should be specified on the command line as

   ```
   test_convergence <integrator> <timestep> <numsteps>
   ```

   as with `ode_solve`. The output should be a single number, the value of the error norm (8). Deliverable: `test_convergence.cc`.

5. In the language of your choice, write a script named `run_tests` to run `test_convergence` for all three integrators, up to time $t = 25$, for timesteps 0.1, 0.01, 0.001, 0.0001, 0.00001. Your script should write three files: `euler_conv.out`, `ab2_conv.out`, and `rk4_conv.out`, which contain the convergence results in the format

   ```
   timestep1  error1
   timestep2  error2
   ...
   ```

   You should see that the explicit Euler scheme is first-order accurate (the error should be proportional to h), while Adams-Bashforth is second-order accurate (h^2), and Runge-Kutta is fourth-order accurate (h^4), at least up to numerical precision of the computer. Deliverable: `run_tests`.

You should also submit a `Makefile` that builds your targets `ode_solve`, and `test_convergence`.

When you are finished, submit these files using the CS Dropbox system at `https://dropbox.cs.princeton.edu/APC524_F2014/CXXIntegrator`.

P.T.O.

Optional registry design

Your `ode_solve.cc` file (as written at our behest) is a little inelegant. Adding a new integrator would require you to edit the main program, adding the integrator's header file and extending the `if ... else if ... else` block. A better solution is illustrated in the provided files `registry_example.cc` and `simpleRegistry.h`.

Write a version of `ode_solve.cc` that does not `#include` any of the files `adams-bashforth.h`, `euler.h`, or `runge-kutta.h`; you'll probably want to cannibalize `simpleRegistry.h` to create `registry.h` (which'll now create `Integrators`, not `Animals` — note that the `Factory` will have a different signature). You'll also need to modify your integrators, so submit the modified `euler.cc` as well, and `registry.h`. Do not submit `adams-bashforth.h`, `euler.h`, or `runge-kutta.h`, as your program should not use them.

The `Registry` uses a few interesting tricks which we'll discuss further in the *Design Patterns* lecture. In the meantime, you might find it helpful to note that:

- The `getInstance` method has a static variable which is constructed the first time the method is called, and retained in memory until the program exits. The private methods (*e.g.* the constructor) are made private to ensure that `getInstance` is the only way to return a `Registry` — and it's always the same instance.

- The `RegisterAnimal` variables are constructed before the `main` routine is entered. Their constructors call `Registry::declare`. There's no reason why these variables should all be in the same file.

- The `makeAnimal` template simply calls the proper constructor, but it's a real function whose address can be registered by the `Registry::declare` call. This sort of function is called a *Factory Function* — hence the typedef.