C++ for scientific computing

Friday Assignment: Meeting a unit test specification

Joe Pitt-Francis

2021 (v1.6)

This assignment description is available online via the address:
http://www.cs.ox.ac.uk/people/joe.pitt-francis/oxfordonly/FridayAssignment.pdf

Background:
The intentions behind setting a short assessed programming assignment for this course are:

- To reinforce some of the concepts you have learnt
- To give you a clear goal to meet which proves you have completed the course
- To give you a chance to learn a few things which you may not have covered in lectures or exercises. Specifically these are:
  - The abstract class pattern
  - Simple use of STL vectors
  - Unit testing (with the CxxTest framework)
  - ODE solver implementation

A nice feature of this assignment is that you are provided with a testing infrastructure which gives an implicit specification of the code you are required to write. This means that when you've written code such that all the tests pass then you will know that you have completed the exercise.

Note: If you need to attempt this exercise from a non-DTC machine you will need to first install CxxTest. This can be done via a Debian package (on Ubuntu) or the Python pip package manager.

Step 1: initial compilation
Make a new Eclipse project and populate it with the contents of this download :
http://www.cs.ox.ac.uk/people/joe.pitt-francis/oxfordonly/FridayAssignment.zip
You have been provided with source two classes (`Exception` and `AbstractOdeSolver`) and with two test files. **You will not need to edit any of these 6 source files.** If you do make edits to them during the course of the exercise then make sure that your final version works with the original versions of these files. There is also a Makefile. Check to see what project build `make all` does at this stage. It should stop after a few lines with “No rule to make target `ForwardEulerOdeSolver.cpp`”.

In order for the code to compile, you need to create the `ForwardEulerOdeSolver` class (as two files, `ForwardEulerOdeSolver.hpp` and `ForwardEulerOdeSolver.cpp`). This class should inherit from `AbstractOdeSolver` as indicated by the figure below, and should provide the one method:

```
void ForwardEulerOdeSolver::Solve()
```

(This method implementation does not have to contain any code yet.) Now check the code compiles.

Step 2: writing a forward Euler solver
At this point building the code with `make all` should not only compile the classes but will also use `cxxtestgen` to build an executable file `TestOdeSolversRunner` from one of the tests, and will then run that executable. The good news is that 50% of the tests in the `TestOdeSolvers` test-suite pass immediately. This is because the test-suite is organised into 4 individual tests. The first two: `TestSomeExamples` which illustrates how tests work and `TestAbstractClassMethods` which tests that all the functionality I have written in the abstract class works, both pass. The other two tests which check `ForwardEulerOdeSolver::Solve()`works correctly are the tests which are failing. Eclipse will highlight which test lines are failures.

If you need a revision tutorial about the Forward Euler method then please ask now. Your task is to implement the `Solve()`method so that it solves from the initial values over a time-interval with a fixed time step using forward Euler. On exit from the method you should have populated `mSolutionTrace` with (x,y) values (as `Pair` structures) and `mTimeTrace` with the corresponding times. See `TestAbstractClassMethods` for testing code examples which illustrate how to populate these vectors. The test-suite uses two simple model ordinary differential 2-d systems of equations. These are

plugged into the solver as right-hand side functions. They are defined at the top of the test-suite file and both have clear analytic solutions.

As you develop your `Solve()`method you should reduce the number of failing lines. One or two of the test lines are harder to satisfy than others. **Attempt to have no failures** but the fewer the better. Please make sure that you have comments in your code.
If at this point you are running out of time then you may submit in order to get partial credit.

Step 3: writing a better solver
Now make an edit to the `Makefile`: remove or comment the top line and switch it for the line below so that `make` will build and run both of the test-suite files:
`all: TestOdeSolversRunner TestHigherOrderOdeSolverRunner`
The new test-suite requires you make another class which should again provide the one method:
`void HigherOrderOdeSolver::Solve()`
It tests the convergence behaviour of the two solvers and checks that the higher-order one is better. My model answer (which passes the tests) uses a standard second order Runge-Kutta method.
Again work on your solution until there are few lines (ideally no lines) of failing tests.

For extra credit:
Extension 1. Use `TestHigherOrderOdeSolver::TestSimpleCircleConvergence()`as the model for plotting convergence behaviour for these two solvers and at least one other. Use your favourite plotting program to make a log-log plot with time-step on the x-axis and error on the y-axis. Save this as a .png file.
Extension 2. Code up the 2-d version of the Van der Pol oscillator as a right-hand side function and find some parameters which give interesting orbits.[1] Use the `DumpToFile()` method and your favourite plotting tool to produce a plot of its behaviour. Save this plot as a .png file.

Deliverables
~~You should email your work to~~ dtcmodules@dtc.ox.ac.uk ~~by 5pm on Friday.~~
This should be as a single email attachment which identifies who you are:
`<your_name>.zip` or `<your_name>.tgz` will be most suitable. This file will contain:

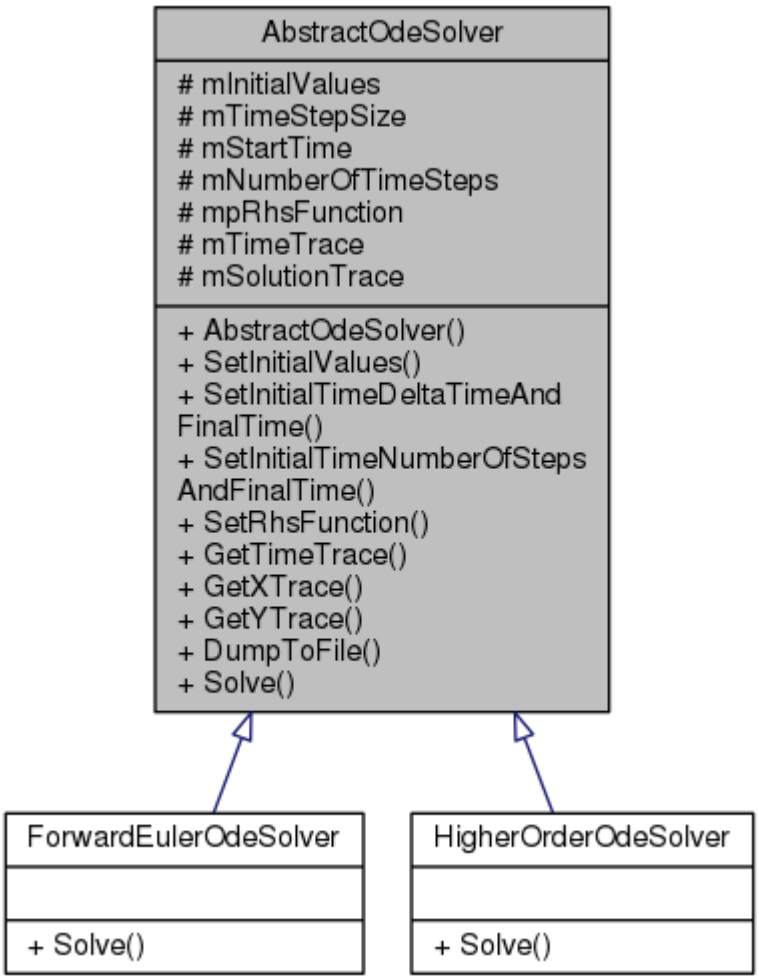
- `ForwardEulerOdeSolver.cpp`
- `ForwardEulerOdeSolver.hpp`
- `HigherOrderOdeSolver.cpp`
- `HigherOrderOdeSolver.hpp`

For **extension 1** add:

- Source for your test-suite testing code
- Source for your extra solver(s)
- A log-log convergence .png plot
- Raw data for your plot (.csv or columns)
- Your new Makefile

For **extension 2** add:

- Your test-suite (with ODE function)
- A phase-plane plot
- Raw data for the plot
- Your new Makefile



1 To see what I mean do an image search for “Van der Pol oscillator” (but don't forget “oscillator”).