InFoMM C++ Skills Training

Practical 4 (Lectures 11-13)

This practical is based on the material in lectures 11-13. It builds on the ODE stepper class that you implemented in Practical 3, using templates to enable it to accept generic state types and derivative functions.

1. Template the ODE stepper class you wrote in the previous practical so that it can use a generic state type `T`. You may assume that this type has an assignment operator `=` defined which can copy one variable to another, and that it has `+` and `*` operators which allow state variables to be added together and multiplied by scalars.

2. Add another template argument `typename F` that represents a class that can calculate the time derivative of your generic vector state type `T` (i.e. this class replaces the functionality of the `dydx` member function you wrote in practical 3). You can assume that the class `F` has a function call operator `operator()` which takes 2 arguments, a `const T&` holding the current state y, and a `T&` argument which holds dy/dx after the operator is called (See the `negx` type below for an example). Make a constructor for your `OdeInt` class that accepts an argument of type `const F&` and saves this in a `private` variable for later use by the `step` function.

(Note: In general, a lightweight class with a function call operator is called a *function object*, or *functor*. A lambda function is an example of a function object. A variable of this class can be passed to different functions/classes and used as a function internally. Function object are used heavily in the C++ STL.)

3. Template your `integrate` function using `typename T` and `typename F` representing the same classes in 1. and 2. Add another argument that takes a functor of type `const F&`, used to calculate the derivative.

4. Use your `integrate` function to integrate $dy/dx = -y$ as before, passing it an instance of the following function object. Verify the accuracy of the result.

```
struct negx {
    void operator()(const double &y, double &dydx) { dydx = -y; }
};
```

5. Write a partial specialisation for your ODE stepper class that uses a `std::vector` for the state type. Note that this type does not have the same assignment or arithmetic operators you probably have been using, so you will need to modify this code in the specialised class. Now repeat step 5. using a `std::vector<double>` for the state type.

6. Now we will integrate a second order ODE $\ddot{y} = -y$, with initial condition $y = 1$, $\dot{y} = 0$ at $x = 10$ and using a step size of $dx = 10^{-3}$. Convert $\ddot{y} = -y$ to the standard form $\frac{d\mathbf{y}}{dx} = f(\mathbf{y})$, where $\mathbf{y}$ is now a vector, and implement an appropriate function object that can calculate the rhs function $f(\mathbf{y})$ using a `std::vector<double>` for your state type `T`. You can check your answer using the analytical solution $y(x) = cos(x)$.

7. Do the same integration using an Eigen vector (i.e. `Eigen::VectorXd`) for the state type `T`, and verify that it gives the same result.

8. Using STL algorithms, perform a convergence study for your integration in Q6. That is, create a `std::vector<double>` of decreasing step sizes and convert this (perhaps using `std::transform`) to another `std::vector<double>` of p-norm distances between the calculated and analytical solutions.

9. Modify your ODE stepper class so that it uses exceptions. You could throw an exception for negative step sizes, or if the state variable becomes infinite or Not-A-Number (see http://en.cppreference.com/w/cpp/numeric/math/isfinite).