Ordinary differential equations

Introduction

Many scientific problems can be formulated in terms of a system of *ordinary differential equations* (ODE),

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}) \ , \tag{1}$$

with an initial condition

$$\mathbf{y}(x_0) = \mathbf{y}_0 \ , \tag{2}$$

where $\mathbf{y}' \equiv d\mathbf{y}/dx$, and the boldface variables $\mathbf{y}$ and $\mathbf{f}(x, \mathbf{y})$ are generally understood as column-vectors.

Runge-Kutta methods

Runge-Kutta methods are one-step methods for numerical integration of ODE (1). The solution $\mathbf{y}$ is advanced from the point x_0 to $x_1 = x_0 + h$ using a one-step formula

$$\mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}, \tag{3}$$

where $\mathbf{y}_1$ is the approximation to $\mathbf{y}(x_1)$, and $\mathbf{k}$ is a cleverly chosen (vector) constant. The Runge-Kutta methods are distinguished by their *order*: a method has order p if it can integrate exactly an ODE where the solution is a polynomyal of order p. In other words, if the error of the method is $O(h^{p+1})$ for small h.

The first order Runge-Kutta method is the Euler's method,

$$\mathbf{k} = \mathbf{f}(x_0, \mathbf{y}_0) \ . \tag{4}$$

Second order Runge-Kutta methods advance the solution by an auxiliary evaluation of the derivative, e.g. the *mid-point method*,

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0) \ , \\ \mathbf{k}_{1/2} &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0) \ , \\ \mathbf{k} &= \mathbf{k}_{1/2} \ , \end{aligned} \tag{5}$$

or the *two-point method*,

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0), \\ \mathbf{k}_1 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_0), \\ \mathbf{k} &= \frac{1}{2}(\mathbf{k}_0 + \mathbf{k}_1) \ . \end{aligned} \tag{6}$$

These two methods can be combined into a third order method,

$$\mathbf{k} = \frac{1}{6}\mathbf{k}_0 + \frac{4}{6}\mathbf{k}_{1/2} + \frac{1}{6}\mathbf{k}_1 \ . \tag{7}$$

The most commont is the fourth-order method, which is called *RK4* or simply *the Runge-Kutta method*,

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0) \ , \\ \mathbf{k}_1 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0) \ , \\ \mathbf{k}_2 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_1) \ , \\ \mathbf{k}_3 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_2) \ , \\ \mathbf{k} &= \tfrac{1}{6}(\mathbf{k}_0 + 2\mathbf{k}_1 + 2\mathbf{k}_2 + \mathbf{k}_3) \ . \end{aligned} \tag{8}$$

Higher order Runge-Kutta methods have been devised, with the most famous being the Runge-Kutta-Fehlberg fourth/fifth order method, *RKF45*, implemented in the renowned `rkf45` Fortran routine.

Multistep methods

Multistep methods try to use the information about the function gathered at the previous steps. They are generally not *self-starting* as there are no previous points at the start of the integration. The first step must be a one-step method, for example a Runge-Kutta step.

A two-step method

Given the previous point, $(x_{-1}, \mathbf{y}_{-1})$, in addition to the current point $(x_0, \mathbf{y}_0)$, the sought function $\mathbf{y}$ can be approximated in the vicinity of the point x_0 as

$$\bar{\mathbf{y}}(x) = \mathbf{y}_0 + \mathbf{y}'_0 \cdot (x - x_0) + \bar{\mathbf{c}} \cdot (x - x_0)^2, \tag{9}$$

where $\mathbf{y}'_0 = \mathbf{f}(x_0, \mathbf{y}_0)$ and the coefficient $\bar{\mathbf{c}}$ is found from the condition $\bar{\mathbf{y}}(x_{-1}) = \mathbf{y}_{-1}$,

$$\bar{\mathbf{c}} = \frac{\mathbf{y}_{-1} - \mathbf{y}_0 + \mathbf{y}'_0 \cdot (x_0 - x_{-1})}{(x_0 - x_{-1})^2}. \tag{10}$$

The value of the function at the next point, $x_1 \equiv x_0 + h$, can now be estimated as $\bar{\mathbf{y}}(x_1)$ from (9).

The error of this second-order two-step stepper can be estimated by a comparison with a first-order Euler's step, which is given by the linear part of (9). The difference $\|\bar{\mathbf{c}}\|h^2$ between the two steppers can thus serve as the error estimate,

$$\text{err} \approx \|\bar{\mathbf{c}}\|h^2 \,. \tag{11}$$

Two-and-half-step method

One can further increase the order of the approximation (9) by adding a third order term,

$$\bar{\bar{\mathbf{y}}}(x) = \bar{\mathbf{y}}(x) + \bar{\bar{\mathbf{d}}} \cdot (x - x_0)^2 (x - x_{-1}) \,. \tag{12}$$

The coefficient $\bar{\bar{\mathbf{d}}}$ can be found from the matching condition at the half-step,

$$\bar{\bar{\mathbf{y}}}'(x_{1/2}) = \mathbf{f}(x_{1/2}, \bar{\mathbf{y}}(x_{1/2})) \equiv \bar{\mathbf{f}}_{1/2} \,, \tag{13}$$

where $x_{1/2} \doteq x_0 + h/2$. This gives

$$\bar{\bar{\mathbf{d}}} = \frac{\bar{\mathbf{f}}_{1/2} - \mathbf{y}'_0 - 2\bar{\mathbf{c}} \cdot (x_{1/2} - x_0)}{2(x_{1/2} - x_0)(x_{1/2} - x_{-1}) + (x_{1/2} - x_0)^2}. \tag{14}$$

The error estimate at the point $x_1 \equiv x_0 + h$ is again given as the difference between the higher and the lower order steppers,

$$\text{err} \approx \|\bar{\bar{\mathbf{y}}}(x_1) - \bar{\mathbf{y}}(x_1)\| \,. \tag{15}$$

Predictor-corrector methods

A predictor-corrector method uses extra iterations to improve the solution. It is an algorithm that proceeds in two steps. First, the predictor step calculates a rough approximation of $\mathbf{y}(x + h)$. Second, the corrector step refines the initial approximation. Aditionally, the corrector step can be repeated in the hope that this achieves an even better approximation to the true solution.

For example, the two-point Runge-Kutta method (6) is as actually a predictor-corrector method, as it first calculates the *prediction* $\tilde{\mathbf{y}}_1$ for $\mathbf{y}(x_1)$,

$$\tilde{\mathbf{y}}_1 = \mathbf{y}_0 + h\mathbf{f}(x_0, \mathbf{y}_0) \,, \tag{16}$$

and then uses this prediction in a *correction* step,

$$\check{\tilde{\mathbf{y}}}_1 = \mathbf{y}_0 + h\frac{1}{2}\left(\mathbf{f}(x_0, \mathbf{y}_0) + \mathbf{f}(x_1, \tilde{\mathbf{y}}_1)\right) \,. \tag{17}$$

A two-step method with correction

Similarly, one can use the two-step approximation (9) as a predictor, and then improve it by one order with a correction step, namely

$$\check{\bar{\mathbf{y}}}(x) = \bar{\mathbf{y}}(x) + \check{\mathbf{d}} \cdot (x - x_0)^2 (x - x_{-1}). \tag{18}$$

The coefficient $\check{\mathbf{d}}$ can be found from the condition $\check{\bar{\mathbf{y}}}'(x_1) = \bar{\mathbf{f}}_1$, where $\bar{\mathbf{f}}_1 = \mathbf{f}(x_1, \bar{\mathbf{y}}(x_1))$,

$$\check{\mathbf{d}} = \frac{\bar{\mathbf{f}}_1 - \mathbf{y}'_0 - 2\bar{\mathbf{c}} \cdot (x_1 - x_0)}{2(x_1 - x_0)(x_1 - x_{-1}) + (x_1 - x_0)^2}. \tag{19}$$

Equation (18) gives a better estimate, $\mathbf{y}_1 = \check{\bar{\mathbf{y}}}(x_1)$, of the sought function at the point x_1.

In this context the formula (9) serves as *predictor*, and (18) as *corrector*. The difference between the two gives an estimate of the error.

Step size control

Error estimate

The error δy of the integration step for a given method can be estimated e.g. by comparing the solutions for a full-step and two half-steps (the *Runge principle*),

$$\delta y \approx \frac{y_{\text{two\_half\_steps}} - y_{\text{full\_step}}}{2^p - 1}, \tag{20}$$

where p is the order of the algorithm used. It is better to pick formulas where the full-step and two half-step calculations share the evaluations of the function $\mathbf{f}(x, \mathbf{y})$.

Another possibility is to make the same step with two methods of different orders, the difference between the solutions providing an estimate of the error.

In a predictor-corrector method the correction itself can serve as the estimate of the error.

Table 1: Runge-Kutta mid-point stepper with error estimate.

```
#include<functional>
#include<armadillo>
using namespace arma;

void rkstep12(
        const std::function<vec(double,vec)> & f,
        double & x0, vec & y0, double & h,
        vec & y1, vec & dy)
{
        vec k0  = f(x0,y0);
        vec k12 = f(x0+h/2,y0+k0*h/2);
        y1 = y0 + k12*h;
        dy = (k0 - k12)*h/2;
}
```

Adaptive step size control

Let *tolerance* τ be the maximal accepted error consistent with the required absolute, δ, and relative, ϵ, accuracies to be achieved in the integration of an ODE,

$$\tau = \epsilon \|\mathbf{y}\| + \delta \,, \tag{21}$$

where $\|\mathbf{y}\|$ is the "norm" of the column-vector $\mathbf{y}$.

Suppose the inegration is done in n steps of size h_i such that $\sum_{i=1}^{n} h_i = b - a$. Under assumption that the errors at the integration steps are random and independent, the step tolerance τ_i for the step i has to scale as the square root of the step size,

$$\tau_i = \tau \sqrt{\frac{h_i}{b - a}}. \tag{22}$$

Then, if the error e_i on the step i is less than the step tolerance, $e_i \le \tau_i$, the total error E will be consistent with the total tolerance τ,

$$E \approx \sqrt{\sum_{i=1}^{n} e_i^2} \le \sqrt{\sum_{i=1}^{n} \tau_i^2} = \tau \sqrt{\sum_{i=1}^{n} \frac{h_i}{b-a}} = \tau \ . \tag{23}$$

In practice one uses the current values of the function $\mathbf{y}$ in the estimate of the tolerance,

$$\tau_i = (\epsilon \|\mathbf{y}_i\| + \delta) \sqrt{\frac{h_i}{b-a}} \tag{24}$$

The step is accepted if the error is smaller than tolerance. The next step-size can be estimated according to the empirical prescription

$$h_{\text{new}} = h_{\text{old}} \times \left(\frac{\tau}{e}\right)^{\text{Power}} \times \text{Safety}, \tag{25}$$

where Power ≈ 0.25, Safety ≈ 0.95. If the error e_i is larger than tolerance τ_i the step is rejected and a new step with the new step size (25) is attempted.

Table 2: An ODE driver with adaptive step size control.

```
#include<functional>
#include<armadillo>
#include<vector>
#include<stdio.h>
#define NORM(x) norm((x),2)
using namespace arma;

void rkdrive(
        const std::function<vec(double,vec)>& f,
        std::vector<double>& xlist, std::vector<vec>& ylist,
        const double b, double h, const double acc, const double eps)
{
        int    n = ylist.front().size();
        double a = xlist.front();
        vec    dy(n), y1(n);
        while (xlist.back() < b)
                {
                double x = xlist.back();
                vec    y = ylist.back();
                if(x+h>b) h=b-x;
                rkstep12(f,x,y,h,y1,dy);
                double err = NORM(dy);
                double tol = (acc + NORM(y1)*eps)*sqrt(h/(b-a));
                if(tol>err)
                        {
                        xlist.push_back(x+h);
                        ylist.push_back(y1);
                        }
                if(err>0) h = h*pow(tol/err,0.25)*0.95;
                else h = 2*h;
                }
}
```