1 Ordinary differential equations

1.1 Introduction

Ordinary differential equations (ODE) are generally defined as differential equations in one variable where the highest order derivative enters linearly. Such equations invariably arise in many different contexts throughout mathematics (and science generally) as soon as changes in the system at hand are considered, usually with respect to variations of certain parameters.

Ordinary differential equations can be generally reformulated as (coupled) systems of first-order ordinary differential equations,

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

where $\mathbf{y}' \doteq d\mathbf{y}/dx$, and the variables $\mathbf{y}$ and the *right-hand side function* $\mathbf{f}(x, \mathbf{y})$ are understood as column-vectors. For example, a second order differential equation in the form

$$u'' = g(x, u, u') \tag{2}$$

can be rewritten as a system of two first-order equations,

$$\begin{cases} y_1' = y_2 \\ y_2' = g(x, y_1, y_2) \end{cases} , \tag{3}$$

using the variable substitution $y_1 = u$, $y_2 = u'$.

In practice ODEs are usually supplemented with boundary conditions which pick out a certain class or a unique solution of the ODE. In the following we shall mostly consider the *initial value problem*: an ODE with the boundary condition in the form of an initial condition at a given point a,

$$\mathbf{y}(a) = \mathbf{y}_0 \ . \tag{4}$$

The problem is then to find the value of the solution $\mathbf{y}$ at some other point b.

Finding a solution to an ODE is often referred to as *integrating* the ODE.

An integration algorithm typically advances the solution from the initial point a to the final point b in a number of discrete steps

$$\{x_0 \doteq a, x_1, \ldots, x_{n-1}, x_n \doteq b\}. \tag{5}$$

An efficient algorithm tries to integrate an ODE using as few steps as possible under the constraint of the given accuracy goal. For this purpose the algorithm should continuously adjust the step-size during the integration, using few larger steps in the regions where the solution is smooth and perhaps many smaller steps in more treacherous regions.

Typically, an adaptive step-size ODE integrator is implemented as two routines. One of them—called *driver*—monitors the local errors and tolerances and adjusts

the step-sizes. To actually perform a step the driver calls a separate routine—the *stepper*—which advances the solution by one step, using one of the many available algorithms, and estimates the local error. The GNU Scientific Library, GSL, implements about a dozen of different steppers and a tunable adaptive driver.

In the following we shall discuss several of the popular driving algorithms and stepping methods for solving initial-value ODE problems.

1.2 Adaptive step-size control

Let *tolerance* τ be the maximal accepted error consistent with the required accuracy to be achieved in the integration of an ODE. Suppose the integration 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 statistically uncorrelated, the local tolerance τ_i for the step i has to scale as the square root of the step-size,

$$\tau_i = \tau \sqrt{\frac{h_i}{b-a}}. \tag{6}$$

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

$$E \approx \sqrt{\sum_{i=1}^{n} e_i^2} \le \sqrt{\sum_{i=1}^{n} \tau_i^2} = \tau \sqrt{\sum_{i=1}^{n} \frac{h_i}{b-a}} = \tau \, . \tag{7}$$

The current step h_i is accepted if the local error e_i is smaller than the local tolerance τ_i, after which the next step is attempted with the step-size adjusted according to the following empirical prescription [2],

$$h_{i+1} = h_i \times \left(\frac{\tau_i}{e_i}\right)^{\text{Power}} \times \text{Safety}, \tag{8}$$

where Power ≈ 0.25 and Safety ≈ 0.95.

If the local error is larger than the local tolerance the step is rejected and a new step is attempted with the step-size adjusted according to the same prescription (8).

One simple prescription for the local tolerance τ_i and the local error e_i to be used in (8) is

$$\tau_i = (\epsilon \|\mathbf{y}_i\| + \delta) \sqrt{\frac{h_i}{b-a}} \, , \; e_i = \|\delta \mathbf{y}_i\| \, , \tag{9}$$

where δ and ϵ are the required absolute and relative precision and $\delta \mathbf{y}_i$ is the estimate of the integration error at the step i.

A more elaborate prescription considers components of the solution separately,

$$(\tau_i)_k = \big(\epsilon |(\mathbf{y}_i)_k| + \delta\big) \sqrt{\frac{h_i}{b-a}} \, , \; (\mathbf{e}_i)_k = |(\delta \mathbf{y}_i)_k| \, , \tag{10}$$

Table 1: An implementation of an ODE driver in Python

```
def driver(f,a,ya,b,h=0.125,acc=0.01,eps=0.01) :
    x=a; y=ya; xlist=[x]; ylist=[y]
    while True :
        if x>=b  : return (xlist,ylist) # return the path
        if x+h>b : h=b-x
        (yh,dy) = stepper(f,x,y,h) # stepper returns y(x+h) and δy
        tol = (acc+eps*yh.norm())*math.sqrt(h/(b-a))
        err = dy.norm()
        if err<tol :
            x+=h; y=yh; xlist.append(x); ylist.append(y)
        if err>0 : h*=min( (tol/err)**0.25*0.95 , 2 )
        else     : h*=2
```

where the index k runs over the components of the solution. In this case the step acceptance criterion also becomes component-wise: the step is accepted, if

$$\forall k : (\mathbf{e}_i)_k < (\tau_i)_k \,. \tag{11}$$

The factor τ_i/e_i in the step adjustment formula (8) is then replaced by

$$\frac{\tau_i}{e_i} \to \min_k \frac{(\tau_i)_k}{(\mathbf{e}_i)_k} \,. \tag{12}$$

Yet another refinement is to include the derivatives $\mathbf{y}'$ of the solution into the local tolerance estimate, either overall,

$$\tau_i = \Big(\epsilon\alpha\|\mathbf{y}_i\| + \epsilon\beta\|\mathbf{y}'_i\| + \delta\Big)\sqrt{\frac{h_i}{b-a}} \,, \tag{13}$$

or component-wise,

$$(\tau_i)_k = \Big(\epsilon\alpha|(\mathbf{y}_i)_k| + \epsilon\beta|(\mathbf{y}'_i)_k| + \delta\Big)\sqrt{\frac{h_i}{b-a}} \,. \tag{14}$$

The weights α and β are chosen by the user.

Table 1 shows an implementation of the discussed driver in Python.

1.3 Error estimate

In an adaptive step-size algorithm the stepping routine must provide an estimate of the integration error, upon which the driver bases its strategy to determine the optimal step-size for a user-specified accuracy goal.

A stepping method is generally characterized by its *order*: a method has order p if it can integrate exactly an ODE where the solution is a polynomial of order p. In other words, for small h the error of the order-p method is $O(h^{p+1})$.

There are two popular methods to estimate the error: step doubling, where one compares the results from the full step and from two half-steps, and two-orders method where one compares the results from steppers of two different orders.

1.3.1 Step doubling (Runge's principle)

For sufficiently small steps the error δy of an integration step for a method of a given order p can be estimated by comparing the solution $\mathbf{y}_{\text{full\_step}}$, obtained with one full-step integration, against a potentially more precise solution, $\mathbf{y}_{\text{two\_half\_steps}}$, obtained with two consecutive half-step integrations,

$$\delta\mathbf{y} = \frac{\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}}}{2^p - 1} \,. \tag{15}$$

where p is the order of the algorithm used. Indeed, if the step-size h is small, we can assume

$$\delta\mathbf{y}_{\text{full\_step}} = Ch^{p+1} \,, \tag{16}$$

$$\delta\mathbf{y}_{\text{two\_half\_steps}} = 2C\left(\frac{h}{2}\right)^{p+1} = \frac{Ch^{p+1}}{2^p} \,, \tag{17}$$

where $\delta\mathbf{y}_{\text{full\_step}}$ and $\delta\mathbf{y}_{\text{two\_half\_steps}}$ are the errors of the full-step and two half-steps integrations, and C is an unknown constant. The two can be combined as

$$\begin{aligned} \mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}} &= \delta\mathbf{y}_{\text{full\_step}} - \delta\mathbf{y}_{\text{two\_half\_steps}} \\ &= \frac{Ch^{p+1}}{2^p}(2^p - 1) \,, \end{aligned} \tag{18}$$

from which it follows that

$$\frac{Ch^{p+1}}{2^p} = \frac{\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}}}{2^p - 1} \,. \tag{19}$$

One has, of course, to take the potentially more precise $\mathbf{y}_{\text{two\_half\_steps}}$ as the approximation to the solution $\mathbf{y}$. Its error is then given as

$$\delta\mathbf{y}_{\text{two\_half\_steps}} = \frac{Ch^{p+1}}{2^p} = \frac{\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}}}{2^p - 1} \,, \tag{20}$$

which had to be demonstrated. This prescription is often referred to as the *Runge's principle*.

One drawback of the Runge's principle is that the full-step and the two half-step calculations generally do not share evaluations of the right-hand side function $\mathbf{f}(x, \mathbf{y})$, and therefore many extra evaluations are needed to estimate the error.

1.3.2 Different orders

An alternative prescription for error estimation is to make the same step-size integration using two methods of *different orders*, with the difference between the two solutions providing the estimate of the error. If the lower order method mostly uses the same evaluations of the right-hand side function—in which case it is called

embedded in the higher order method—the error estimate does not need additional evaluations.

Predictor-corrector methods are naturally of embedded type: the correction—which generally increases the order of the method—itself can serve as the estimate of the error.

1.4 Runge-Kutta steppers

The Runge-Kutta steppers approximate the solution of the differential equation using polynomials. The coefficients of the polynomials are determined by sampling the right-hand-side, $\mathbf{f}(x, \mathbf{y})$, of the differential equation at certain points (mostly) within the step.

1.4.1 Linear stepper

The linear approximation is given by the first two terms of the Taylor expansion of the solution at the point x_i,

$$\mathbf{y}(x) \approx \mathbf{p}_1(x) = \mathbf{y}_i + \mathbf{y}'_i(x - x_i) \,, \tag{21}$$

where the derivative $\mathbf{y}'_i$ is determined from the equation itself, $\mathbf{y}'_i = \mathbf{f}(x, \mathbf{y}_i)$. In this parlance the values of the derivative are often denoted as $\mathbf{k}$ with index, for example the derivative $\mathbf{y}'_i$ is often called $\mathbf{k}_0$,

$$\mathbf{p}_1(x) = \mathbf{y}_i + \mathbf{k}_0(x - x_i) \,. \tag{22}$$

With this notation the linear Runge-Kutta stepper (called the Euler's rule) is given as

$$\mathbf{k}_0 = \mathbf{f}(x, \mathbf{y}_i) \tag{23}$$

$$\mathbf{y}_{i+1} \approx \mathbf{p}_1(x + h) = \mathbf{y}_i + \mathbf{k}_0 h \,. \tag{24}$$

1.4.2 Quadratic steppers and Butcher tableaux

One can improve upon the $\mathbf{p}_1$-approximation by adding a second order term,

$$\mathbf{y}(x) \approx \mathbf{p}_2(x) = \mathbf{p}_1(x) + \mathbf{c}(x - x_i)^2 \,, \tag{25}$$

where the coefficient $\mathbf{c}$ can be found by matching the polynomial $\mathbf{p}_2$ against our differential equation at a certain point z within the step,

$$\mathbf{p}'_2(z) = \mathbf{f}(z, \mathbf{p}_1(z)) \,. \tag{26}$$

This condition should be viewed in the sense of Taylor expansion, that is why we used $\mathbf{p}_1$ polynomial at the right-hand-side. This gives

$$\mathbf{c} = \frac{\mathbf{f}(z, \mathbf{p}_1(z)) - \mathbf{p}'_1(z)}{2(z - x_i)} \,. \tag{27}$$

One usually identifies the sampling point z by the corresponding fraction of the step, $z = x_i + \alpha h$. Written with the α the **c**-coefficient is given as

$$\mathbf{c} = \frac{\mathbf{f}(x_i + \alpha h, \mathbf{y}_i + \alpha \mathbf{k}_0 h)) - \mathbf{k}_0}{2\alpha h} = \frac{\mathbf{k}_1 - \mathbf{k}_0}{2\alpha h} \ , \tag{28}$$

where

$$\mathbf{k}_1 \doteq \mathbf{f}(x_i + \alpha h, \mathbf{y}_i + \alpha \mathbf{k}_0 h) \ . \tag{29}$$

Finally, the second order approximating polynomial is given as

$$\mathbf{p}_2(x) = \mathbf{y}_i + \mathbf{k}_0 (x - x_i) + (\mathbf{k}_1 - \mathbf{k}_0) \frac{(x - x_i)^2}{2\alpha h} \ . \tag{30}$$

The value of the function at the end of the step, $\mathbf{y}_{i+1} \approx \mathbf{p}_2(x_i + h)$, is then given as

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \left(1 - \frac{1}{2\alpha}\right) \mathbf{k}_0 h + \frac{1}{2\alpha} \mathbf{k}_1 h \ . \tag{31}$$

Summarizing, the (generic) second order Runge-Kutta stepper is given as

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_i, \mathbf{y}_i) && (32) \\ \mathbf{k}_1 &= \mathbf{f}(x_i + \alpha h, \mathbf{y}_i + \alpha \mathbf{k}_0 h) && (33) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \left(1 - \frac{1}{2\alpha}\right) \mathbf{k}_0 h + \frac{1}{2\alpha} \mathbf{k}_1 h \ . && (34) \end{aligned}$$

It is customary in this business to represent the Runge-Kutta steppers with the so called Butcher tableaux which collect the step-sizes and the coefficient of the polynomials in a table. For example, the Butcher tableau of out generic second order stepper is given as

	step sizes	coefficients before $\mathbf{k}_i$	
$\mathbf{k}_0$	0		
$\mathbf{k}_1$	α	α	
$\mathbf{y}_{i+1}$		$(1 - \frac{1}{2\alpha})$	$\frac{1}{2\alpha}$

(35)

The intermediate step α can be chosen arbitrarily (or from a certain condition) the stepper is of second order for any α. Following is a list with the most popular choices of α.

- Midpoint method

0		
1/2	1/2	
	0	1

(36)

- Heun's method

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array} \tag{37}$$

- Ralston's method

$$\begin{array}{c|cc} 0 & & \\ 2/3 & 2/3 & \\ \hline & 1/4 & 3/4 \end{array} \tag{38}$$

1.4.3 Cubic approximation

One can further increase the order of the approximating polynomial,

$$\mathbf{p}_3(x) = \mathbf{p}_2(x) + \mathbf{d}(x - x_i)^3 \,, \tag{39}$$

where d can be found from the matching condition at a (possibly) different point z_2 within the step,

$$\mathbf{p}_3'(z_2) = \mathbf{f}(z_2, \mathbf{p}_2(z_2)) \,. \tag{40}$$

This gives

$$\mathbf{d} = \frac{\mathbf{f}(z_2, \mathbf{p}_2(z_2)) - \mathbf{p}_2'(z_2)}{3(z_2 - x_i)^2} \,. \tag{41}$$

Taking $z = x_i + \beta h$ and denoting

$$\mathbf{k}_2 \doteq \mathbf{f}\left(z_2, \mathbf{p}_2(z_2)\right) = \mathbf{f}\left(x_i + \beta h, \mathbf{y}_i + \left(\beta - \frac{\beta^2}{2\alpha}\right)\mathbf{k}_0 h + \frac{\beta^2}{2\alpha}\mathbf{k}_1 h\right) \tag{42}$$

we get

$$\mathbf{d} = \frac{\mathbf{k}_2 - \left(\mathbf{k}_0 + (\mathbf{k}_1 - \mathbf{k}_0)\,\frac{\beta}{\alpha}\right)}{3\beta^2 h^2} \,, \tag{43}$$

and

$$\begin{aligned} \mathbf{p}_3(x) = \mathbf{y}_i + \mathbf{k}_0(x - x_i) + \frac{\mathbf{k}_1 - \mathbf{k}_0}{2\alpha h}(x - x_i)^2 \\ + \frac{\mathbf{k}_2 - \left(\mathbf{k}_0 + (\mathbf{k}_1 - \mathbf{k}_0)\,\frac{\beta}{\alpha}\right)}{3\beta^2 h^2}(x - x_i)^3 \,. \end{aligned} \tag{44}$$

This gives the following (generic) cubic Runge-Kutta stepper,

$$\begin{aligned}
\mathbf{k}_0 &= \mathbf{f}(x_i, \mathbf{y}_i) && (45)\\
\mathbf{k}_1 &= \mathbf{f}(x_i + \alpha h, \mathbf{y}_i + \alpha \mathbf{k}_0 h) && (46)\\
\mathbf{k}_2 &= \mathbf{f}\left(x_i + \beta h, \mathbf{y}_i + \left(\beta - \frac{\beta^2}{2\alpha}\right)\mathbf{k}_0 h + \frac{\beta^2}{2\alpha}\mathbf{k}_1 h\right) && (47)\\
\mathbf{y}_{i+1} &= \mathbf{y}_i + \left(1 - \frac{1}{2\alpha} - \frac{1}{3\beta^2} + \frac{1}{3\alpha\beta}\right)\mathbf{k}_0 h + \left(\frac{1}{2\alpha} - \frac{1}{3\alpha\beta}\right)\mathbf{k}_1 h \\
&\quad + \left(\frac{1}{3\beta^2}\right)\mathbf{k}_1 h\ . && (48)
\end{aligned}$$

The corresponding Butcher tableau is

$$\begin{array}{c|ccc}
0 & & & \\
\hline
\alpha & \alpha & & \\
\hline
\beta & \beta - \frac{\beta^2}{2\alpha} & \frac{\beta^2}{2\alpha} & \\
\hline
 & 1 - \frac{1}{2\alpha} - \frac{1}{3\beta^2} + \frac{1}{3\alpha\beta} & \frac{1}{2\alpha} - \frac{1}{3\alpha\beta} & \frac{1}{3\beta^2}
\end{array} \qquad (49)$$

The intermediates steps α and β can be chosen largely arbitrarily. Following is a list with several popular cubic Runge-Kutta steppers that follow the formulae from our generic stepper.

- Heun's third-order method

$$\begin{array}{c|ccc}
0 & & & \\
1/3 & 1/3 & & \\
2/3 & 0 & 2/3 & \\
\hline
 & 1/4 & 0 & 3/4
\end{array} \qquad (50)$$

- Ralston's third-order method

$$\begin{array}{c|ccc}
0 & & & \\
1/2 & 1/2 & & \\
3/4 & 0 & 3/4 & \\
\hline
 & 2/9 & 1/3 & 4/3
\end{array} \qquad (51)$$

- 8/15th third-order method

$$\begin{array}{c|cccc} 0 & & & & \\ 8/15 & 8/15 & & & \\ 2/3 & 1/4 & 5/12 & & \\ \hline & 1/4 & 0 & 3/4 & \end{array} \tag{52}$$

And, in principle, the coefficients in the rows can be also reshuffled somewhat (preferable shifting the weight to the right) with the condition that the sum of the coefficients in the row must be equal one.

1.4.4 Higher order steppers

Using this approach one can easily further increase the order of the approximating polynomial by adding higher powers of $(x - x_i)$ and fixing the coefficients in front by an additional sampling of the right-hand-side at certain points within the step. This led to a bunch of steppers of different orders with different sampling points. For example, here is the classic Runge-Kutta fourth-order method,

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \quad . \tag{53}$$

One can find a list of different Runge-Kutta methods in Wikipedia.

1.4.5 Embedded methods with error estimates

The embedded Runge-Kutta methods in addition to advancing the solution by one step also produce an estimate of the local error of the step. This is done by having two methods in the tableau, one with a certain order p and another one with order $p-1$. The difference between the two methods gives the estimate of the local error. If the lower order method uses the same **k**-values as the higher order method, it is called *embedded*. Embedded methods allow effective estimate of the error without extra evaluations of the right-hand-side.

Since the embedded rule uses the same **k**'s the Butcher's tableau for this kind of method is simply extended by one row to give the coefficients of the lower order rule.

Table 2: Embedded midpoint/Euler method with error estimate

```
public static (vector,vector) rkstep12
   (Func<double,vector,vector> f, double x, vector y, double h)
{
vector k0 = f(x,y);                  /* embedded lower order formula (Euler) */
vector k1 = f(x+h/2,y+k0*(h/2));/* higher order formula (midpoint) */
vector yh = y+k1*h;                  /* y(x+h) estimate */
vector dy = (k1-k0)*h;               /* error estimate */
return (yh,dy);
}
```

The simplest embedded methods are Heun-Euler method,

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \\ & 1 & 0 \end{array}, \tag{54}$$

and midpoint-Euler method,

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \\ & 1 & 0 \end{array}, \tag{55}$$

which both combine methods of orders 2 and 1. Table (2) shows a C# implementation of the embedded midpoint/Euler method with error estimate.

Here is a simple embedded method of orders 2 and 3,

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 3/4 & 0 & 3/4 & \\ \hline & 2/9 & 3/9 & 4/9 \\ & 0 & 1 & 0 \end{array}, \tag{56}$$

The *Bogacki-Shampine method* [1] combines methods of orders 3 and 2,

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 3/4 & 0 & 3/4 & & \\ 1 & 2/9 & 1/3 & 4/9 & \\ \hline & 2/9 & 1/3 & 4/9 & 0 \\ & 7/24 & 1/4 & 1/3 & 1/8 \end{array} \quad . \tag{57}$$

Bogacki and Shampine argue that their method has better stability properties and actually outperforms higher order methods at lower accuracy goal calculations. This method has the FSAL—First Same As Last—property: the value $\mathbf{k}_4$ at one step equals $\mathbf{k}_1$ at the next step; thus only three function evaluations are needed per step.

The Runge-Kutta-Fehlberg method [3]—called *RKF45*—implemented in the renowned `rkf45` Fortran routine, has two methods of orders 5 and 4:

0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	−7200/2197	7296/2197			
1	439/216	−8	3680/513	−845/4104		
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	
	16/135	0	6656/12825	28561/56430	−9/50	2/55
	25/216	0	1408/2565	2197/4104	−1/5	0

1.5 Implicit methods

Instead of the forward Euler method one could employ the backward Euler method where the derivative is approximated as

$$y'(x) \approx \frac{y(x) - y(x-h)}{h} \ , \tag{58}$$

which gives the following (backward Euler) stepper,

$$y_{x+h} = y_x + hf(x+h, y_{x+h}) \ . \tag{59}$$

The backward Euler methods is an *implicit* method: one has to solve the above equation to find y_{x+h}. It generally costs time to solve this equation numerically – a disadvantage as compared to explicit methods. However implicit methods are

usually more stable for stiff (difficult) equations where a larger step h can be used as compared to explicit methods.

Just like with explicit methods one can devise higher-order implicit methods, for example, the implicit Heun's method (trapezoidal rule),

$$y_{x+h} = y_x + h\frac{1}{2}\big(f(x, y_x) + f(x+h, y_{x+h})\big) \ . \tag{60}$$

1.6 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 steps at the start of the integration. The first step must be done with a one-step method like Runge-Kutta.

A number of multistep methods have been devised (and named after different mathematicians); we shall only consider a few simple ones here to get the idea of how it works.

1.6.1 Two-step method

Given the previous point, $(x_{i-1}, \mathbf{y}_{i-1})$, in addition to the current point $(x_i, \mathbf{y}_i)$, the sought function $\mathbf{y}$ can be approximated in the vicinity of the point x_i as a second order polynomial,

$$\mathbf{y}(x) \approx \mathbf{p}_2(x) = \mathbf{y}_i + \mathbf{y}'_i \cdot (x - x_i) + \mathbf{c} \cdot (x - x_i)^2, \tag{61}$$

where $\mathbf{y}'_i = \mathbf{f}(x_i, \mathbf{y}_i)$ and the coefficient $\mathbf{c}$ can be found from the condition

$$\mathbf{p}_2(x_{i-1}) = \mathbf{y}_{i-1} \ , \tag{62}$$

which gives

$$\mathbf{c} = \frac{\mathbf{y}_{i-1} - \mathbf{y}_i + \mathbf{y}'_i \cdot (x_i - x_{i-1})}{(x_i - x_{i-1})^2} \ . \tag{63}$$

The value $\mathbf{y}_{i+1}$ of the function at the next point, $x_{i+1} \doteq x_i + h$, can now be estimated as $\mathbf{y}_{i+1} = \mathbf{p}_2(x_{i+1})$ from (61).

The error of this second-order two-step stepper can be estimated by a comparison with the first-order Euler's step, which is given by the linear part of (61). The correction term $\mathbf{c}h^2$ can serve as the error estimate,

$$\delta\mathbf{y} = \mathbf{c}h^2 \ . \tag{64}$$

1.6.2 Two-step method with extra evaluation

One can further increase the order of the approximation (61) by adding a third order term,

$$\mathbf{y}(x) \approx \mathbf{p}_3(x) = \mathbf{p}_2(x) + \mathbf{d} \cdot (x - x_i)^2(x - x_{i-1}) \ . \tag{65}$$

The coefficient $\mathbf{d}$ can be found from the matching condition at a certain point t inside the interval,

$$\mathbf{p}_3'(t) = \mathbf{f}(t, \mathbf{p}_2(t)) \,, \tag{66}$$

where $x_i < t < x_i + h$. This gives

$$\mathbf{d} = \frac{\mathbf{f}(t, \mathbf{p}_2(t)) - \mathbf{y}_i' - 2\mathbf{c} \cdot (t - x_i)}{2(t - x_i)(t - x_{i-1}) + (t - x_i)^2}. \tag{67}$$

The error estimate at the point $x_{i+1} \doteq x_0 + h$ is again given as the difference between the higher and the lower order methods,

$$\delta\mathbf{y} = \mathbf{p}_3(x_{i+1}) - \mathbf{p}_2(x_{i+1}) \,. \tag{68}$$

1.7 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. Additionally 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 (**??**) is as actually a predictor-corrector method, as it first calculates the *prediction* $\tilde{\mathbf{y}}_{i+1}$ for $\mathbf{y}(x_{i+1})$,

$$\tilde{\mathbf{y}}_{i+1} = \mathbf{y}_i + h\mathbf{f}(x_i, \mathbf{y}_i) \,, \tag{69}$$

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

$$\check{\mathbf{y}}_{i+1} = \mathbf{y}_i + h\frac{1}{2}\left(\mathbf{f}(x_i, \mathbf{y}_i) + \mathbf{f}(x_{i+1}, \tilde{\mathbf{y}}_{i+1})\right) \,. \tag{70}$$

1.7.1 Two-step method with correction

Similarly, one can use the two-step approximation (61) as a predictor, and then improve it by one order with a correction step, namely

$$\check{\mathbf{y}}(x) = \bar{\mathbf{y}}(x) + \check{\mathbf{d}} \cdot (x - x_i)^2(x - x_{i-1}). \tag{71}$$

The coefficient $\check{\mathbf{d}}$ can be found from the condition $\check{\mathbf{y}}'(x_{i+1}) = \bar{\mathbf{f}}_{i+1}$, where $\bar{\mathbf{f}}_{i+1} \doteq \mathbf{f}(x_{i+1}, \bar{\mathbf{y}}(x_{i+1}))$,

$$\check{\mathbf{d}} = \frac{\bar{\mathbf{f}}_{i+1} - \mathbf{y}_i' - 2\mathbf{c} \cdot (x_{i+1} - x_i)}{2(x_{i+1} - x_i)(x_{i+1} - x_{i-1}) + (x_{i+1} - x_i)^2}. \tag{72}$$

Equation (71) gives a better estimate, $\mathbf{y}_{i+1} = \check{\mathbf{y}}(x_{i+1})$, of the sought function at the point x_{i+1}. In this context the formula (61) serves as *predictor*, and (71) as *corrector*. The difference between the two gives an estimate of the error.

This method is equivalent to the two-step method with an extra evaluation where the extra evaluation is done at the full step.

References

[1] Przemyslaw Bogacki and Lawrence F. Shampine. A 3(2) pair of Runge–Kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989.

[2] M. Galassi et al. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 3rd edition, 2009.

[3] Erwin Fehlberg. *Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems*. NASA Technical Report, 1969.