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

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

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 algorthm 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 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 $O(h^{p+1})$.

1.2.1 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 $\mathbf{y}_{\rm full\_step}$, obtained with one full-step integration, against a potentially more precise solution, $\mathbf{y}_{\rm two\_half\_steps}$, obtained with two consecutive half-step integrations,

$$\delta\mathbf{y} = \frac{\mathbf{y}_{\rm full\_step} - \mathbf{y}_{\rm two\_half\_steps}}{2^p - 1} . \tag{6}$$

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

$$\delta\mathbf{y}_{\rm full\_step} = Ch^{p+1} , \tag{7}$$

$$\delta\mathbf{y}_{\rm two\_half\_steps} = 2C\left(\frac{h}{2}\right)^{p+1} = \frac{Ch^{p+1}}{2^p} , \tag{8}$$

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

$$\begin{aligned} \mathbf{y}_{\rm full\_step} - \mathbf{y}_{\rm two\_half\_steps} &= \delta\mathbf{y}_{\rm full\_step} - \delta\mathbf{y}_{\rm two\_half\_steps} \\ &= \frac{Ch^{p+1}}{2^p}(2^p - 1) , \end{aligned} \tag{9}$$

from which it follows that

$$\frac{Ch^{p+1}}{2^p} = \frac{\mathbf{y}_{\rm full\_step} - \mathbf{y}_{\rm two\_half\_steps}}{2^p - 1} . \tag{10}$$

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

$$\delta\mathbf{y}_{\rm two\_half\_steps} = \frac{Ch^{p+1}}{2^p} = \frac{\mathbf{y}_{\rm full\_step} - \mathbf{y}_{\rm two\_half\_steps}}{2^p - 1} , \tag{11}$$

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.

1.2.2 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.

1.3 Runge-Kutta methods

Runge-Kutta methods are one-step methods which advance the solution over the current step using only the information gathered from withing the step itself. The solution $\mathbf{y}$ is advanced from the point x_i to $x_{i+1} = x_i + h$, where h is the step-size, using a one-step formula,

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{k}, \tag{12}$$

where $\mathbf{y}_{i+1}$ is the approximation to $\mathbf{y}(x_{i+1})$, and the value $\mathbf{k}$ is chosen such that the method integrates exactly an ODE whose solution is a polynomial of the highest possible order.

The Runge-Kutta methods are distinguished by their order. Again, a method has order p if it can integrate exactly an ODE where the solution is a polynomial of order p (or if for small h the error of the method is $O(h^{p+1})$).

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

$$\mathbf{k} = \mathbf{f}(x_0, \mathbf{y}_0)\,. \tag{13}$$

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

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0)\,, \\ \mathbf{k}_{1/2} &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0)\,, \\ \mathbf{k} &= \mathbf{k}_{1/2}\,, \end{aligned} \tag{14}$$

or the *two-point method*, also called the *Heun's method*

$$
\begin{aligned}
\mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0), \\
\mathbf{k}_1 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_0), \\
\mathbf{k} &= \frac{1}{2}(\mathbf{k}_0 + \mathbf{k}_1) \ .
\end{aligned} \tag{15}
$$

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

$$
\mathbf{k} = \frac{1}{6}\mathbf{k}_0 + \frac{4}{6}\mathbf{k}_{1/2} + \frac{1}{6}\mathbf{k}_1 \ . \tag{16}
$$

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

$$
\begin{aligned}
\mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0) \ , \\
\mathbf{k}_1 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0) \ , \\
\mathbf{k}_2 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_1) \ , \\
\mathbf{k}_3 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_2) \ , \\
\mathbf{k} &= \tfrac{1}{6}\mathbf{k}_0 + \tfrac{1}{3}\mathbf{k}_1 + \tfrac{1}{3}\mathbf{k}_2 + \tfrac{1}{6}\mathbf{k}_3 \ .
\end{aligned} \tag{17}
$$

A general Runge-Kutta method can be written as

$$
\mathbf{y}_{n+1} = \mathbf{y}_n + \sum_{i=1}^{s} b_i h \mathbf{k}_i \ , \tag{18}
$$

where

$$
\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(x_n, \mathbf{y}_n) \ , \\
\mathbf{k}_2 &= \mathbf{f}(x_n + c_2 h, \mathbf{y}_n + a_{21} h\mathbf{k}_1) \ , \\
\mathbf{k}_3 &= \mathbf{f}(x_n + c_3 h, \mathbf{y}_n + a_{31} h\mathbf{k}_1 + a_{32} h\mathbf{k}_2) \ , \\
&\ \vdots \\
\mathbf{k}_s &= \mathbf{f}(x_n + c_s h, \mathbf{y}_n + a_{s1} h\mathbf{k}_1 + a_{s2} h\mathbf{k}_2 + \cdots + a_{s,s-1} h\mathbf{k}_{s-1}) \ .
\end{aligned} \tag{19}
$$

To specify a particular Runge-Kutta method one needs to provide the coefficients $\{a_{ij} | 1 \le j < i \le s\}$, $\{b_i | i = 1..s\}$ and $\{c_i | i = 1..s\}$. The matrix $[a_{ij}]$ is called the Runge-Kutta matrix, while the coefficients b_i and c_i are known as the weights and the nodes. These data are usually arranged in the so called *Butcher's tableau*,

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array} \ . \tag{20}
$$

For example, the Butcher's tableau for the RK4 method is

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} . \tag{21}$$

1.3.1 Embeded 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 bitween the two methods gives the the estimate of the local error. The lower order method is *embedded* in the higher order method, that is, it uses the same **k**-values. This allows a very effective estimate of the error.

The embedded lower order method is written as

$$\mathbf{y}^*_{n+1} = \mathbf{y}_n + \sum_{i=1}^{s} b^*_i h \mathbf{k}_i \,, \tag{22}$$

where $\mathbf{k}_i$ are the same as for the higher order method. The error estimate is then given as

$$\mathbf{e}_n = \mathbf{y}_{n+1} - \mathbf{y}^*_{n+1} = \sum_{i=1}^{s} (b_i - b^*_i) h \mathbf{k}_i \,. \tag{23}$$

The Butcher's tableau for this kind of method is extended by one row to give the values of b^*_i.

The simplest embedded methods are Heun-Euler method,

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \\ & 1 & 0 \end{array} , \tag{24}$$

and midpoint-Euler method,

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \\ & 1 & 0 \end{array} , \tag{25}$$

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

Table 1: Embedded midpoint/Euler method with error estimate

```
void rkstep12(void f(int n, double x, double*yx, double*dydx),
int n, double x, double* yx, double h, double* yh, double* dy){
  int i; double k0[n],yt[n],k12[n]; /* VLA: gcc -std=c99 */
  f(n,x    ,yx,k0);  for(i=0;i<n;i++) yt[i]=yx[i]+ k0[i]*h/2;
  f(n,x+h/2,yt,k12); for(i=0;i<n;i++) yh[i]=yx[i]+k12[i]*h;
  for(i=0;i<n;i++) dy[i]=(k0[i]-k12[i])*h/2; /* optimistic */
}
```

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

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 3/4 & 0 & 3/4 & \\ \hline & 2/9 & 3/9 & 4/9 \\ & 0 & 1 & 0 \end{array} , \tag{26}$$

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

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 3/4 & 0 & 3/4 & & \\ 1 & 2/9 & 1/3 & 4/9 & \\ \hline & 2/9 & 1/3 & 4/9 & 0 \\ & 7/24 & 1/4 & 1/3 & 1/8 \end{array} . \tag{27}$$

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. Following is a simple implementation which does not utilise this property for the sake of presentational clarity.

```
void rkstep23( void f(int n,double x,double* y,double* dydx),
int n, double x, double* yx, double h, double* yh, double* dy){
   int i; double k1[n],k2[n],k3[n],k4[n],yt[n]; /* VLA: -std=c99 */
   f(n,    x      ,yx,k1); for(i=0;i<n;i++) yt[i]=yx[i]+1./2*k1[i]*h;
   f(n,x+1./2*h,yt,k2); for(i=0;i<n;i++) yt[i]=yx[i]+3./4*k2[i]*h;
   f(n,x+3./4*h,yt,k3); for(i=0;i<n;i++)
      yh[i]=yx[i]+(2./9  *k1[i]+1./3*k2[i]+4./9*k3[i])*h;
   f(n,   x+h     ,yh,k4);   for(i=0;i<n;i++){
      yt[i]=yx[i]+(7./24*k1[i]+1./4*k2[i]+1./3*k3[i]+1./8*k4[i])*h;
      dy[i]=yh[i]-yt[i];
      }
}
```

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

1.4 Implicit methods

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

$$y'(x) \approx \frac{y(x) - y(x-h)}{h} \,, \tag{28}$$

which gives the following (backward Euler) stepper,

$$y_{x+h} = y_x + hf(x+h, y_{x+h}) \,. \tag{29}$$

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 expicit 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 examle, the trapezoidal rule,

$$y_{x+h} = y_x + h\frac{1}{2}\left(f(x, y_x) + f(x+h, y_{x+h})\right) \,. \tag{30}$$

1.5 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.

1.5.1 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,

$$\mathbf{y}(x) \approx \mathbf{p}_2(x) = \mathbf{y}_i + \mathbf{y}'_i \cdot (x - x_i) + \mathbf{c} \cdot (x - x_i)^2, \tag{31}$$

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

$$\mathbf{p}_2(x_{i-1}) = \mathbf{y}_{i-1} \ , \tag{32}$$

which gives

$$\mathbf{c} = \frac{\mathbf{y}_{i-1} - \mathbf{y}_i + \mathbf{y}'_i \cdot (x_i - x_{i-1})}{(x_i - x_{i-1})^2} \ . \tag{33}$$

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 (31).

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 (31). The correction term $\mathbf{c}h^2$ can serve as the error estimate,

$$\delta\mathbf{y} = \mathbf{c}h^2 \ . \tag{34}$$

1.5.2 Two-step method with extra evaluation

One can further increase the order of the approximation (31) by adding a third order term,

$$\mathbf{y}(x) \approx \mathbf{p}_3(x) = \mathbf{p}_2(x) + \mathbf{d} \cdot (x - x_i)^2 (x - x_{i-1}) \ . \tag{35}$$

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

$$\mathbf{p}'_3(t) = \mathbf{f}(t, \mathbf{p}_2(t)) \doteq \mathbf{f}_2 \ , \tag{36}$$

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

$$\mathbf{d} = \frac{\mathbf{f}_2 - \mathbf{y}'_i - 2\mathbf{c} \cdot (t - x_i)}{2(t - x_i)(t - x_{i-1}) + (t - x_i)^2} . \tag{37}$$

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,

$$\delta\mathbf{y} = \mathbf{p}_3(x_{i+1}) - \mathbf{p}_2(x_{i+1}) \ . \tag{38}$$

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

$$\tilde{\mathbf{y}}_{i+1} = \mathbf{y}_i + h\mathbf{f}(x_i, \mathbf{y}_i) \;, \tag{39}$$

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

$$\check{\tilde{\mathbf{y}}}_{i+1} = \mathbf{y}_i + h\frac{1}{2}\left(\mathbf{f}(x_i, \mathbf{y}_i) + \mathbf{f}(x_{i+1}, \tilde{\mathbf{y}}_{i+1})\right) \;. \tag{40}$$

1.6.1 Two-step method with correction

Similarly, one can use the two-step approximation (31) as a predictor, and then improve it by one order with a correction step, namely

$$\check{\bar{\mathbf{y}}}(x) = \bar{\mathbf{y}}(x) + \check{\mathbf{d}} \cdot (x - x_i)^2(x - x_{i-1}). \tag{41}$$

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

$$\check{\mathbf{d}} = \frac{\bar{\mathbf{f}}_{i+1} - \mathbf{y}'_i - 2\mathbf{c} \cdot (x_{i+1} - x_i)}{2(x_{i+1} - x_i)(x_{i+1} - x_{i-1}) + (x_{i+1} - x_i)^2}. \tag{42}$$

Equation (41) gives a better estimate, $\mathbf{y}_{i+1} = \check{\bar{\mathbf{y}}}(x_{i+1})$, of the sought function at the point x_{i+1}. In this context the formula (31) serves as *predictor*, and (41) 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.

1.7 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 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 statistically uncorrelated, the local tolerance τ_i for the step i has to scale as the square root of the step-size,

$$\tau_i = \tau\sqrt{\frac{h_i}{b-a}}. \tag{43}$$

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,

$$E \approx \sqrt{\sum_{i=1}^{n} e_i^2} \le \sqrt{\sum_{i=1}^{n} \tau_i^2} = \tau\sqrt{\sum_{i=1}^{n} \frac{h_i}{b-a}} = \tau \;. \tag{44}$$

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],

$$h_{i+1} = h_i \times \left(\frac{\tau_i}{e_i}\right)^{\text{Power}} \times \text{Safety}, \tag{45}$$

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 (45).

One simple prescription for the local tolerance τ_i and the local error e_i to be used in (45) is

$$\tau_i = (\epsilon\|\mathbf{y}_i\| + \delta)\sqrt{\frac{h_i}{b-a}}\ ,\ e_i = \|\delta\mathbf{y}_i\|\ , \tag{46}$$

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,

$$(\tau_i)_k = \big(\epsilon|(\mathbf{y}_i)_k| + \delta\big)\sqrt{\frac{h_i}{b-a}}\ ,\ (\mathbf{e}_i)_k = |(\delta\mathbf{y}_i)_k|\ , \tag{47}$$

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

$$\forall k:\ (\mathbf{e}_i)_k < (\tau_i)_k\ . \tag{48}$$

The factor τ_i/e_i in the step adjustment formula (45) is then replaced by

$$\frac{\tau_i}{e_i} \rightarrow \min_k \frac{(\tau_i)_k}{(\mathbf{e}_i)_k}\ . \tag{49}$$

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

$$\tau_i = \Big(\epsilon\alpha\|\mathbf{y}_i\| + \epsilon\beta\|\mathbf{y}'_i\| + \delta\Big)\sqrt{\frac{h_i}{b-a}}\ , \tag{50}$$

or commponent-wise,

$$(\tau_i)_k = \Big(\epsilon\alpha|(\mathbf{y}_i)_k| + \epsilon\beta|(\mathbf{y}'_i)_k| + \delta\Big)\sqrt{\frac{h_i}{b-a}}\ . \tag{51}$$

The weights α and β are chosen by the user.

Following is a simple C-language implementation of the described algorithm.

```
int ode_driver(void f(int n,float x,float*y,float*dydx),
int n,float*xlist ,float**ylist ,
float b,float h,float acc,float eps,int max){
  int i,k=0; float x,*y,s,err,normy,tol,a=xlist[0],yh[n],dy[n];
  while(xlist[k]<b){
    x=xlist[k], y=ylist[k]; if(x+h>b) h=b-x;
    ode_stepper(f,n,x,y,h,yh,dy);
    s=0; for(i=0;i<n;i++) s+=dy[i]*dy[i]; err  =sqrt(s);
    s=0; for(i=0;i<n;i++) s+=yh[i]*yh[i]; normy=sqrt(s);
    tol=(normy*eps+acc)*sqrt(h/(b-a));
    if(err<tol){ /* accept step and continue */
      k++; if(k>max-1) return -k; /* uups */
      xlist[k]=x+h; for(i=0;i<n;i++)ylist[k][i]=yh[i];
      }
    if(err>0) h*=pow(tol/err,0.25)*0.95; else h*=2;
    } /* end while */
  return k+1; } /* return the number of entries in xlist/ylist */
```

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.