5 Error Control

5.1 The Milne Device and Predictor-Corrector Methods

We already discussed the basic idea of the predictor-corrector approach in Section 2. In particular, there we gave the following algorithm that made use of the 2nd-order AB and AM (trapezoid) methods.

Algorithm

$\tilde{\boldsymbol{y}}_{n+2} = \boldsymbol{y}_{n+1} + \frac{h}{2}\left[3\boldsymbol{f}(t_{n+1}, \boldsymbol{y}_{n+1}) - \boldsymbol{f}(t_n, \boldsymbol{y}_n)\right]$

$\boldsymbol{y}_{n+2} = \boldsymbol{y}_{n+1} + \frac{h}{2}\left[\boldsymbol{f}(t_{n+1}, \boldsymbol{y}_{n+1}) + \boldsymbol{f}(t_{n+2}, \tilde{\boldsymbol{y}}_{n+2})\right]$

$\kappa = \frac{1}{6}|\tilde{\boldsymbol{y}}_{n+2} - \boldsymbol{y}_{n+2}|$

if κ is relatively large, then

$h \leftarrow h/2$ (i.e., reduce the stepsize)

repeat

else if κ is relatively small, then

$h \leftarrow 2h$ (i.e., increase the stepsize)

else

continue (i.e., keep h)

end

The basic idea of this algorithm is captured in a framework known as the *Milne device* (see the flowchart on p.77 of the Iserles book). Earlier we explained how we arrived at the formula for the estimate, κ, of the local truncation error in the special case above.

For a general pair of explicit AB and implicit AM methods of (the same) order p we have local truncation errors of the form

$$\boldsymbol{y}(t_{n+s}) - \tilde{\boldsymbol{y}}_{n+s} = \tilde{c}h^{p+1}\boldsymbol{y}^{(p+1)}(\eta_{AB}) \tag{49}$$

$$\boldsymbol{y}(t_{n+s}) - \boldsymbol{y}_{n+s} = ch^{p+1}\boldsymbol{y}^{(p+1)}(\eta_{AM}). \tag{50}$$

Note that (as in the earlier case of 2nd-order methods) we have shifted the indices for the two methods so that they align, i.e., the value to be determined at the current time step has subscript $n+s$.

If we assume that the derivative $\boldsymbol{y}^{(p+1)}$ is nearly constant over the interval of interest, i.e., $\boldsymbol{y}^{(p+1)}(\eta_{AB}) \approx \boldsymbol{y}^{(p+1)}(\eta_{AM}) \approx \boldsymbol{y}^{(p+1)}(\eta)$, then we can subtract equation (50) from equation (49) to get

$$\boldsymbol{y}_{n+s} - \tilde{\boldsymbol{y}}_{n+s} \approx (\tilde{c} - c)h^{p+1}\boldsymbol{y}^{(p+1)}(\eta),$$

and therefore

$$h^{p+1}\boldsymbol{y}^{(p+1)}(\eta) \approx \frac{1}{\tilde{c} - c}\left(\boldsymbol{y}_{n+s} - \tilde{\boldsymbol{y}}_{n+s}\right).$$

If we substitute this expression back into (50) we get an estimate for the error at this time step as

$$\|\boldsymbol{y}(t_{n+2})-\boldsymbol{y}_{n+2}\| = |c|h^{p+1}\|\boldsymbol{y}^{(p+1)}(\eta_{AM})\| \approx |c|h^{p+1}\|\boldsymbol{y}^{(p+1)}(\eta)\| \approx \left|\frac{c}{\tilde{c}-c}\right| \|\boldsymbol{y}_{n+s}-\tilde{\boldsymbol{y}}_{n+s}\|.$$

Thus, in the flowchart, the *Milne estimator* κ is of the form

$$\kappa = \left|\frac{c}{\tilde{c}-c}\right| \|\boldsymbol{y}_{n+s}-\tilde{\boldsymbol{y}}_{n+s}\|.$$

The reason for the upper bound $h\delta$ on κ for the purpose of stepsize adjustments (instead of simply a tolerance δ for the global error) is motivated by the heuristics that the transition from the local truncation error to the global error reduces the local error by an order of h.

Remark 1. We point out that the use of the (explicit) predictor serves two purposes here. First, it eliminates the use of iterative methods to cope with the implicitness of the corrector, and secondly — for the same price — we also get an estimator for the local error that allows us to use variable stepsizes, and thus compute the solution more efficiently. Sometimes the error estimate κ is even added as a correction (or extrapolation) to the numerical solution $\boldsymbol{y}_{n+s}$. However, this process rests on a somewhat shaky theoretical foundation, since one cannot guarantee that the resulting value really is more accurate.

2. As mentioned earlier, since an s-step Adams method requires startup values at equally spaced points, it may be necessary to compute these values via polynomial interpolation (see the "remeshing" steps in the flowchart).

5.2 Richardson Extrapolation

Another simple, but rather inefficient, way to estimate the local error κ (and then again use the general framework of the flowchart to obtain an adaptive algorithm) is to look at two numerical approximations coming from the *same* method: one based on a single step with stepsize h, and the other based on two steps with stepsize $h/2$. We first describe the general idea of *Richardson extrapolation*, and then illustrate the idea on the example of Euler's method.

Whenever we approximate a quantity F by a numerical approximation scheme F_h with a formula of the type

$$F = F_h + \underbrace{\mathcal{O}(h^p)}_{=E_h}, \qquad p \geq 1, \tag{51}$$

we can use an extrapolation method to combine already computed values (at stepsizes h and $h/2$) to obtain a better estimate.

Assume we have computed two approximate values F_h (using stepsize h) and $F_{h/2}$ (using 2 steps with stepsize $h/2$) for the desired quantity F. Then the error for the stepsize $h/2$ satisfies

$$E_{h/2} \approx c\left(\frac{h}{2}\right)^p = c\frac{h^p}{2^p} \approx \frac{1}{2^p}E_h.$$

Therefore, using (51),

$$F - F_{h/2} = E_{h/2} \approx \frac{1}{2^p} E_h = \frac{1}{2^p}(F - F_h).$$

This implies

$$F\left(1 - \frac{1}{2^p}\right) \approx F_{h/2} - \frac{1}{2^p} F_h$$

or

$$F \approx \frac{2^p}{2^p - 1}\left[F_{h/2} - \frac{F_h}{2^p}\right].$$

The latter can be rewritten as

$$F \approx \frac{2^p}{2^p - 1} F_{h/2} - \frac{1}{2^p - 1} F_h. \tag{52}$$

This is the *Richardson extrapolation* formula, a weighted average of $F_{h/2}$ and F_h.

Since the error E_h is given by $F - F_h$, and F in turn can be approximated via (52) we also obtain an *estimate for the error* E_h, namely

$$E_h = F - F_h \approx \frac{2^p}{2^p - 1} F_{h/2} - \frac{1}{2^p - 1} F_h - F_h = \frac{2^p}{2^p - 1}\left[F_{h/2} - F_h\right].$$

We can use this in place of κ and obtain an adaptive algorithm following the flowchart.

Example We know that Euler's method

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + h\boldsymbol{f}(t_n, \boldsymbol{y}_n)$$

produces a solution that is accurate up to terms of order $\mathcal{O}(h)$ so that $p = 1$. If we denote by $\boldsymbol{y}_{n+1,h}$ the solution obtained taking one step with step size h from t_n to t_{n+1}, and by $\boldsymbol{y}_{n+1,h/2}$ the solution obtained taking two steps with step size h from t_n to t_{n+1}, then we can use

$$\boldsymbol{y}_{n+1} = 2\boldsymbol{y}_{n+1,h/2} - \boldsymbol{y}_{n+1,h}$$

to improve the accuracy of Euler's method, or to obtain the error estimate

$$\kappa = \|\boldsymbol{y}_{n+1,h/2} - \boldsymbol{y}_{n+1,h}\|.$$

5.3 Embedded Runge-Kutta Methods

For (explicit) Runge-Kutta methods another strategy exists for obtaining adaptive solvers: the so-called *embedded Runge-Kutta methods*. With an embedded Runge-Kutta method we also compute the value $\boldsymbol{y}_{n+1}$ twice. However, it turns out that we can design methods of *different orders* that use the *same* function evaluations, i.e., the function evaluations used for a certain lower-order method are embedded in a second higher-order method.

In order to see what the local error estimate κ looks like we assume we have the (lower order) method that produces a solution $\boldsymbol{y}_{n+1}$ such that

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}(t_{n+1}) + \boldsymbol{c}h^{p+1} + \mathcal{O}(h^{p+2}),$$

where $\boldsymbol{y}$ is the exact (local) solution based on the initial condition $\boldsymbol{y}(t_n) = \boldsymbol{y}_n$. Similarly, the higher-order method produces a solution $\tilde{\boldsymbol{y}}_{n+1}$ such that

$$\tilde{\boldsymbol{y}}_{n+1} = \boldsymbol{y}(t_{n+1}) + \mathcal{O}(h^{p+2}).$$

Subtraction of the second of these equations from the first yields

$$\boldsymbol{y}_{n+1} - \tilde{\boldsymbol{y}}_{n+1} \approx \boldsymbol{c}h^{p+1},$$

which is a decent approximation of the error of the lower-order method. Therefore, we have

$$\kappa = \|\boldsymbol{y}_{n+1} - \tilde{\boldsymbol{y}}_{n+1}\|.$$

Example One of the simplest examples of an embedded Runge-Kutta method is the following second-third-order scheme defined by its (combined) Butcher tableaux

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{2}{3} & \frac{2}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & \frac{3}{4} & 0 \\ \hline & \frac{1}{4} & \frac{3}{8} & \frac{3}{8} \end{array}.$$

This implies that the second-order method is given by

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + h\left[\frac{1}{4}\boldsymbol{k}_1 + \frac{3}{4}\boldsymbol{k}_2\right]$$

with

$$\begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(t_n, \boldsymbol{y}_n) \\ \boldsymbol{k}_2 &= \boldsymbol{f}(t_n + \frac{2}{3}h, \boldsymbol{y}_n + \frac{2}{3}h\boldsymbol{k}_1), \end{aligned}$$

and the third-order method looks like

$$\tilde{\boldsymbol{y}}_{n+1} = \boldsymbol{y}_n + h\left[\frac{1}{4}\boldsymbol{k}_1 + \frac{3}{8}\boldsymbol{k}_2 + \frac{3}{8}\boldsymbol{k}_3\right]$$

with the *same* $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$ and

$$\boldsymbol{k}_3 = \boldsymbol{f}(t_n + \frac{2}{3}h, \boldsymbol{y}_n + \frac{2}{3}h\boldsymbol{k}_2).$$

Now, the local error estimator is given by

$$\begin{aligned} \kappa &= \|\boldsymbol{y}_{n+1} - \tilde{\boldsymbol{y}}_{n+1}\| \\ &= \frac{3}{8}h\|\boldsymbol{k}_2 - \boldsymbol{k}_3\|. \end{aligned}$$

If we again use the adaptive strategy outlined in the flowchart, then we have an *adaptive* second-order Runge-Kutta method that uses only three function evaluations per time step.

Remark 1. Sometimes people use the higher-order solution $\tilde{\boldsymbol{y}}_{n+1}$ as their numerical approximation "justified" by the argument that this solution is obtained with a higher-order method. However, a higher-order method need not be more accurate than a lower-order method.

2. Another example of a second-third-order embedded Runge-Kutta method is implemented in MATLAB as `ode23`. However, its definition is more complicated since the third-order method uses the final computed value of the second-order method as its initial slope.

Example We now describe the classical fourth-fifth-order Runge-Kutta-Fehlberg method which was first published in 1970.

The fourth-order method is an "inefficient" one that uses five function evaluations at each time step. Specifically,

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + h\left[\frac{25}{216}\boldsymbol{k}_1 + \frac{1408}{2565}\boldsymbol{k}_3 + \frac{2197}{4104}\boldsymbol{k}_4 - \frac{1}{5}\boldsymbol{k}_5\right]$$

with

$$\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{f}(t_n, \boldsymbol{y}_n), \\
\boldsymbol{k}_2 &= \boldsymbol{f}\left(t_n + \frac{h}{4}, \boldsymbol{y}_n + \frac{h}{4}\boldsymbol{k}_1\right), \\
\boldsymbol{k}_3 &= \boldsymbol{f}\left(t_n + \frac{3}{8}h, \boldsymbol{y}_n + \frac{3h}{32}\boldsymbol{k}_1 + \frac{9h}{32}\boldsymbol{k}_2\right), \\
\boldsymbol{k}_4 &= \boldsymbol{f}\left(t_n + \frac{12}{13}h, \boldsymbol{y}_n + \frac{1932h}{2197}\boldsymbol{k}_1 - \frac{7200h}{2197}\boldsymbol{k}_2 + \frac{7296h}{2197}\boldsymbol{k}_3\right), \\
\boldsymbol{k}_5 &= \boldsymbol{f}\left(t + h, \boldsymbol{y}_n + \frac{439h}{216}\boldsymbol{k}_1 - 8h\boldsymbol{k}_2 + \frac{3680h}{513}\boldsymbol{k}_3 - \frac{845h}{4104}\boldsymbol{k}_4\right).
\end{aligned}$$

The associated six-stage fifth-order method is given by

$$\tilde{\boldsymbol{y}}_{n+1} = \boldsymbol{y}_n + h\left[\frac{16}{135}\boldsymbol{k}_1 + \frac{6656}{12825}\boldsymbol{k}_3 + \frac{28561}{56430}\boldsymbol{k}_4 - \frac{9}{50}\boldsymbol{k}_5 + \frac{2}{55}\boldsymbol{k}_6\right],$$

where $\boldsymbol{k}_1$–$\boldsymbol{k}_5$ are the same as for the fourth-order method above, and

$$\boldsymbol{k}_6 = \boldsymbol{f}\left(t_n + \frac{h}{2}, \boldsymbol{y}_n - \frac{8h}{27}\boldsymbol{k}_1 + 2h\boldsymbol{k}_2 - \frac{3544h}{2565}\boldsymbol{k}_3 + \frac{1859h}{4104}\boldsymbol{k}_4 - \frac{11h}{40}\boldsymbol{k}_5\right).$$

The local truncation error is again estimated by computing the deviation of the fourth-order solution from the fifth-order result, i.e.,

$$\kappa = \|\boldsymbol{y}_{n+1} - \tilde{\boldsymbol{y}}_{n+1}\|.$$

The coefficients of the two methods can be listed in a combined Butcher tableaux (see Table 5).

The advantage of this embedded method is that an adaptive fifth-order method has been constructed that uses only six function evaluations for each time step.

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

Table 5: Combined Butcher tableaux for the fourth-fifth-order Runge-Kutta-Fehlberg method.

Remark 1. Other embedded fourth-fifth order methods exist. Nearly every software package has an implementation of (at least) one of them. In Maple the default numerical solver for the `dsolve` command is the RKF45 method. The function `ode45` in MATLAB uses as 4-5 pair by Dormand and Prince. The default fourth-fifth order method in Mathematica uses a pair by Bogacki and Shampine.

2. Other embedded Runge-Kutta methods also exist. For example, one of the first fifth-sixth-order methods is due to Dormand and Prince (1980). The coefficients of this method can be found in the Kincaid/Cheney textbook.

3. As mentioned earlier, Runge-Kutta methods are quite popular. This is probably due to the fact that they have been known for a long time, and are relatively easy to program. However, for *stiff* problems, i.e., problems whose solution exhibits both slow and fast variations in time, we saw earlier (in the MATLAB script `StiffDemo2.m`) that explicit Runge-Kutta methods become very inefficient.