1 Lecture 09: 25 September 2001

Today we will begin the discussion of software issues such as step size control (Sects. II.3 and II.4 of [HNW93], Chap. 7 of [Sha94], and Sect. 5.10 of [Lam91]), continuous output ([HNW93] Sect. II.5) and automatic stiffness detection ([HNW96] Sect. IV.2) schemes for embedded Runge–Kutta methods.

2 Stepsize control

Suppose we take a code such as `rksuite` and try to reverse engineer it to extract the RAS of the methods it implements. Consider running it on the equation $y' = \lambda y$ and seeing when $u_n \to 0$; the code requires an error tolerance ϵ, not too large, and varies h to provide a solution with local error ϵ in each step. When $h\lambda$ is outside the RAS, the solution will be inaccurate and the code will reduce h to maintain accuracy. Thus the numerical solution will always $\to 0$ for $\Re\lambda < 0$, and the determination of the RAS is a little more subtle. The RAS could be determined by finding the cost of the method for each λ: in the RAS, the step size is $O(\epsilon^{-1/p})$ as the error $\epsilon \to 0$, while outside the RAS the step size is determined by stiffness to be $O(1/|\lambda|)$. This is tricky to decide, so we need to know a little more about how such codes choose the step size to control the error.

The goal of step size selection strategies is to produce a solution which is very likely to be within a user–specified tolerance ϵ of the exact solution, at minimal cost (necessarily depending on ϵ like $O(\epsilon^{-1/p})$ for an order–p method). The classical step–doubling approach based on Richardson extrapolation is instructive and will be useful, but modern codes use an alternate approach called "embedded" Runge–Kutta methods. It is important to control the error because engineering computations are worth a lot more when they include error estimates or bounds.

The overall approach to controlling the error of a one–step method of order p is straightforward. Suppose we just took a step of size H from u_{n-1} to u_n and computed an error estimate E_n, and now we want to determine the largest step size h that will give us u_{n+1} with error estimate $E_{n+1} \le \epsilon$. The step h may also be required to satisfy other constraints such as: a singularity occurs at internal time t_*, where we would like to stop and restart, or the user has requested output at certain times T_j. The latter requirement is typically handled either by Hermite interpolation or by continuous Runge–Kutta formulas which we will discuss later.

Given the order p of the method, we know that the local error in one step of size h is $C_p h^{p+1}$ where C_p depends on the solution and order-p derivatives of the right–hand side. Thus a reasonable choice for the new step size would be

$$h = H(\epsilon/E_n)^{1/(p+1)}.$$

This reduces the step size if the error was too large and increases the step if the error was smaller than the tolerance. Of course, in practice we don't want to push our luck and will take

$$h = FH(\epsilon/E_n)^{1/(p+1)}$$

where $F = 0.8$ is a safety factor. Furthermore, we are suspicious of very large step size increases or decreases so we'll probably put

$$h = H \min(5, \max(0.5, F(\epsilon/E_n)^{1/(p+1)}))$$

to be even further on the safe side.

Finally, we make an acceptance/rejection decision based on H and ϵ: if $E_n > \epsilon$, the step $t_{n-1} \to t_n$ is rejected and redone with the new step size h. (We could not do this earlier because we didn't have a better alternative step size.) Otherwise, the step $t_{n-1} \to t_n$ is accepted and the step size h is used to advance from t_n to t_{n+1}. A new error estimate E_{n+1} is computed and the process is repeated until the final time T is reached. We now need to find some systematic way of computing an error estimate E_n at each step.

Exercise 1 *Study the documentation and code for* `rksuite` *to determine its step control strategy and describe it concisely.*

3 Step doubling

The simplest approach to error estimation involves repeating the computation twice with different step sizes and using the more accurate solution to estimate the error in the other (or vice versa).

Starting from the solution u_n, one step of size h commits local error

$$u_{n+1} - y_{n+1} = C_p h^{p+1} + O(h^{p+2}).$$

We could also compute a different numerical solution $\hat{u}_{n+1}$ by taking two steps of size $h/2$; the first step commits error $C_p(h/2)^{p+1}$, which is transported (unchanged to first–order) and added to a similar error $(C_p + O(h))(h/2)^{p+1}$ from the second half–step to get total error

$$\hat{u}_{n+1} - y_{n+1} = 2C_p(h/2)^{p+1} + O(h^{p+2}) = 2^{-p}C_p h^{p+1} + O(h^{p+2}).$$

Taking an appropriate linear combination gives a order–$(p+1)$ accurate result U and an error estimate E_{n+1}:

$$U = \hat{u}_{n+1} - E_{n+1}$$

where

$$E_{n+1} = -\frac{\hat{u}_{n+1} - u_{n+1}}{2^p - 1} = C_p h^{p+1} + O(h^{p+2}).$$

This process is the first step in the famous "Richardson extrapolation" procedure which also provides "Romberg integration" for the calculation of integrals by the extrapolated trapezoidal rule.

This step doubling procedure is one of the oldest error estimates, but is rather expensive. Counting function evaluations alone, an s–stage ERK uses $3s-1$ stages since k_1 can be reused once. Thus the error estimate costs twice as much as the computed solution, which seems overkill for such a tiny (but important) thing. The next error estimate method is more efficient, and is almost universally used in practical codes.

4 Embedded methods

The idea of embedded methods is to use two Runge–Kutta methods of adjacent orders p and $p+1$ to estimate the error in the lower-order method. This sounds even more expensive than step doubling, but the trick is to design the methods so that they share the vector c and matrix A in the Butcher array. Thus we compute the stages k_q only once, and take two different linear combinations to get two solutions and estimate the error. These methods are often presented in an extended Butcher array

$$\begin{array}{c|c} c & A \\ \hline & b^T \\ \hline & \hat{b}^T \\ \hline & e^T \end{array}$$

where the coefficients $e_q = b_q - \hat{b}_q$ give the error estimate

$$E_n = h \sum_{q=1}^{s} e_q k_q$$

from the stages k_q.

Example: We can build two different two–stage ERKs, explicit Euler and the trapezoid rule with Euler predictor, with the same stages

$$\begin{aligned} k_1 &= f(u_n) \\ k_2 &= f(u_n + hk_1). \end{aligned}$$

They produce numerical solutions

$$u_{n+1} = u_n + hk_1$$

and

$$\hat{u}_{n+1} = u_n + h(\frac{1}{2}k_1 + \frac{1}{2}k_2)$$

so we can build an estimate for the error in the explicit Euler solution u_{n+1} by subtraction:

$$E_{n+1} = u_{n+1} - \hat{u}_{n+1} = h(\frac{1}{2}k_1 - \frac{1}{2}k_2) = C_1 h^2 + O(h^3).$$

The extended Butcher array is

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline b^T & 1 & 0 \\ \hline \hat{b}^T & \frac{1}{2} & \frac{1}{2} \\ \hline e^T & \frac{1}{2} & -\frac{1}{2} \end{array}$$

This trivial example has a useful property known as “first same as last” or FSAL. The last row of the matrix A is the same as the row vector b^T, so one stage need not be recomputed in ongoing computations. Here, the second stage in time step $n \to n+1$ is $k_2 = f(u_n + hk_1) = f(u_{n+1})$ so in the next time step $n+1 \to n+2$, the first stage $k_1 = f(u_{n+1})$ is already computed. Thus the method provides an error estimate for Euler's method at no additional cost. It is called Runge–Kutta–Fehlberg $(1, 2)$, and belongs to an extensive family of methods developed in the 1950's by Fehlberg. Other RKF methods include RKF$(2, 3)$, RKF$(4, 5)$, and so forth.

Example: A 3-stage 3rd-order method due to Kutta can be combined with the two-stage explicit midpoint rule to yield

$$\begin{array}{c|ccc}
0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 & 0 \\
1 & -1 & 2 & 0 \\
\hline
b^T & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\
\hline
\hat{b}^T & 0 & 1 & 0 \\
\hline
e^T & -\frac{1}{6} & \frac{1}{3} & -\frac{1}{2}
\end{array}$$

The resulting error estimate is

$$E_n = h(-\frac{1}{6}k_1 + \frac{1}{3}k_2 - \frac{1}{6}k_3) = O(h^3).$$

5 Local extrapolation

Any effective error estimate (as opposed to a bound) poses a dilemma: do we use it as a reasonably rigourous error estimate, or do we subtract it from the computed solution to obtain a more accurate solution without any estimate of the error? Almost all standard codes yield to the temptation to have their cake and eat it too, by adopting the more accurate solution but declaring the error estimate to remain valid. This is reasonable as $h \to 0$, but may give slightly wrong results for the fairly large step sizes used in practical computations. This approach is dignified with the name of "local extrapolation".

During the 1980's, Dormand and Prince reexamined the RKF idea with the aid of computer algebra systems and improved it considerably in two respects. One is related to local extrapolation: Fehlberg optimized the error coefficients for the lower–order method, but almost everyone uses the higher–order method to advance the computation. Dormand-Prince derived methods of orders $(5, 4)$ and $(8, 7)$ (the ordering of p and $p+1$ indicates that the higher–order scheme $p + 1$ is used to advance the solution) with coefficients chosen to minimize the errors in the higher–order computation. These schemes also save one stage (function evaluation) because they have the FSAL property. Numerical results in [HNW93] indicate that Dormand–Prince (5,4) and $(8, 7)$ are essentially the best available Runge–Kutta schemes of moderate fixed order, and often beat multistep methods for ODE systems of moderate size.

Only for extremely large systems of ODEs (such as PDEs discretized by the method of lines) does multistep achieve comparable efficiency.

Exercise 2 *Download the C or Fortran code* dopri5 *and driver routine* dr_dopri5 *from* http://www.unige.ch/math/folks/hairer/software.html *(one of the authors of our textbook [HNW93]). Measure the time (or function evaluations if you prefer) which this code —and each method in* rksuite*— require to solve the Arenstorf problem in* dr_dopri5 *to accuracies ranging from* 10^{-3} *to* 10^{-12}*. Check that your results agree with the order of each method you deduced in previous exercises. Explain how* dopri5 *chooses the step size.*

6 Dense output

A high–order Runge–Kutta method may take fairly large steps, and the user may want to know the solution at fairly small time intervals. One can force the step control mechanism to take small steps, but this wastes tremendous effort and accumulates excessive roundoff error. (Recall that numerical solution of ODEs is always a battle between truncation error which decreases like $O(h^p)$ and roundoff error which grows like $O(h^{-1})$ at worst or $O(h^{-1/2})$ statistically.) Thus practical codes like rksuite or dopri5 usually include some provision for interpolating between the time steps at little additional cost.

One dense output method is to apply any standard interpolation technique to the known solution values u_{n+1}, u_n, $\ldots$. For example, one can linearly interpolate

$$u(t_n + \theta h) = (1 - \theta)u_n + \theta u_{n+1}$$

to $O(h^2)$ accuracy. However, ERK methods permit much higher accuracy with cubic Hermite interpolation, because the solution derivatives $u'_n = f(t_n, u_n)$ and $u'_{n+1} = f(t_{n+1}, u_{n+1})$ are available at the beginning and end of each step. Thus we can approximate

$$u(t_n + \theta h) = p_0(\theta)u_n + q_0(\theta)f(t_n, u_n) + p_1(\theta)u_{n+1} + q_1(\theta)f(t_{n+1}, u_{n+1})$$

where p_i and q_i are cubic polynomials in θ chosen to make this formula exact for all cubic polynomials. This gives $O(h^4)$ accuracy, which suffices for many low–order Runge–Kutta methods.

For high–order ERK methods, a different approach is usually used: take advantage of the many function and derivative approximations inside the step by the stages $k_i = f(t_n + c_i h, g_i)$ where $g_i = u_n + h\sum_j a_{ij}k_j$. One derives "continuous Runge–Kutta formulas" by expanding the error

$$y(t_n + \theta h) - y(t_n) - h(b_1(\theta)k_1 + \cdots + b_s(\theta)k_s)$$

in Taylor series, just as we did when deriving the Runge–Kutta method. But here we let $\theta \in [0, 1]$ be arbitrary, and the b coefficients become functions of θ. We can usually allow one order less accuracy for $\theta < 1$, because the interpolated values will not contribute to the usual accumulation of error as the computation progresses.

An exception which requires the continuous output formula to be designed with full precision over the whole range $0 \le \theta \le 1$ is the solution of delay differential equations such as

$$y' = f(t, y(t), y(t - a(t)))$$

from given values $\{y(t), 0 \le t \le a(0)\}$. A standard approach to such problems, which occur in population dynamics ($a(t) = 9$ months) and epidemiology ($a(t) = 7$ days or so, possibly varying with t), is to apply a continuous order–p Runge–Kutta method with order–p interpolation to provide the back values $y(t - a(t))$. Such methods can be analyzed for stability by seeking exponential solutions and examining their exponents (see Sect. II.15 of [HNW93]).

Exercise 3 *Analyze the 7–stage Dormand–Prince(5,4) method implemented in* `dopri5` *to obtain the coefficients $b_i(\theta)$ of the continuous output formula*

$$u(t_n + \theta h) = u_n + h(b_1(\theta)k_1 + \cdots + b_7(\theta)k_7).$$

(Hint: examine the construction and use of `rcont5`*.)*

7 Automatic stiffness detection

Since ERK methods are extremely expensive for stiff problems (by definition of stiffness), they are often equipped with automatic stiffness detectors which warn the user when excessive expense is being incurred due to stiffness. The usual criterion for warning is that too many steps have been taken and an approximation ρ to the Lipschitz constant L or dominant eigenvalue λ of the Jacobian Df satisfies the condition that $h\rho$ is near the boundary of the region of absolute stability. The main issue is how to obtain ρ. The code `rksuite` has a complicated approach based on the nonlinear power method with many switches and safeguards.

The power method approximates the dominant eigenvalue $\lambda_{\max}$ of a matrix M by multiplying a starting vector v repeatedly by M. If $M = S\Lambda S^{-1}$ is diagonalizable, with eigenvalues satisfying $|\lambda_1| < \cdots << |\lambda_d| = |\lambda_{\max}|$, then k steps of the power method gives a vector

$$M^k v = \sum_{j=1}^{d} v_j \lambda_j^k u_j$$

which is extremely rich in the dominant eigenvector $u_{\max}$ (except in the unlikely case that $v_{\max}$ vanishes exactly). It works extremely well when a single dominant eigenvalue is well separated from the others in magnitude, but less well in general situations. In nonlinear situations, it is often used to estimate a dominant eigenvalue of the Jacobian Df without computing the Jacobian explicitly, by computing a difference

$$\frac{\|f(u) - f(v)\|}{\|u - v\|} \approx Df(z)(u - v) \tag{1}$$

along some vector $u - v$ rich in the dominant eigenvector of Df.

The less sophisticated code `dopri5` (based on the 7–stage Dormand–Prince(5,4) method "DP(5,4)") has a simple two–line stiffness detector which works almost as well most of the time:

$$\rho = \frac{\|k_7 - k_6\|}{\|g_7 - g_6\|}. \tag{2}$$

The problem is declared stiff if a user–supplied limit on the number of steps or function evaluations is exceeded, and if the cost is due to the product $h\rho$ falling near to the boundary of the region of absolute stability: $h\rho \approx 3.25$. Here -3.25 is the left–hand limit of the RAS for DP(5,4), which has stages $k_i = f(t_n + c_i h, u_n + h \sum_j a_{ij} k_j) = f(t_n + c_i h, g_i)$.

Note that Runge–Kutta methods can always be expressed in terms of either the derivative approximations $k_i \approx y'(t_n + c_i h)$ or the function approximations $g_i \approx y(t_n + c_i h)$ with equal generality. In terms of the g variables, a Runge–Kutta method reads

$$\begin{aligned}
g_1 &= u_n + h(a_{11} f(t_n + c_1 h, g_1) + \cdots + a_{1s} f(t_n + c_s h, g_s)) \\
g_2 &= u_n + h(a_{21} f(t_n + c_1 h, g_1) + \cdots + a_{2s} f(t_n + c_s h, g_s)) \\
&\cdots \\
g_s &= u_n + h(a_{s1} f(t_n + c_1 h, g_1) + \cdots + a_{ss} f(t_n + c_s h, g_s)) \\
u_{n+1} &= u_n + h(b_1 f(t_n + c_1 h, g_1 + \cdots b_s f(t_n + c_s h, g_s)),
\end{aligned}$$

so the k variables give a more concise set of formulas. A key feature of the DP(5,4) method is that $c_7 = c_6 = 1$, so stages g_6 and g_7 provide two separate approximations of the solution at time t_{n+1}. Also the method is "First Same As Last" (FSAL); the last stage g_7 is the same as u_{n+1} because the last row of A is the same as the 5th order approximation given by b^T.

Two facts make Eq. (2) an effective stiffness detector: first, both g_7 and g_6 are fourth–order accurate approximations to u_{n+1}, so

$$g_7 - g_6 = O(h^4).$$

This can be verified by an arduous Taylor expansion. Second, $g_7 - g_6$ is a good approximation to the dominant eigenvector of the Jacobian Df. To see this, consider a linear problem $y' = Jy$ where we can pretend J is a large negative number for the time being. Writing the method as

$$g = u_n e + hJAg$$

(where e is the vector of all 1's) and solving for g gives

$$g = (I - hJA)^{-1} u_n e = (I + hJA + (hJA)^2 + \cdots + (hJA)^6) u_n e.$$

Here we used the geometric series and the fact that A is a strictly lower triangular 7×7 matrix, so $A^7 = 0$. Subtracting the last two stages in the vector g shows the second fact, that $g_7 - g_6$ is some degree–6 polynomial in the matrix hJ, times u_n. A degree–6 polynomial which is $O(h^4)$ must be of the form $g_7 - g_6 = C_4(hJ)^4 + O(hJ)^5$, so it looks very much like the result of four steps of the power method applied to the starting vector u_n. Thus this stiffness detector (2) has the form of Eq. (1), along a vector $g_7 - g_6$ rich in the dominant eigenvector. In practice, it works extremely well and reliably signals when to switch to an IRK method.

References

[HNW93] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential equations I : Nonstiff problems.* Springer-Verlag, second revised edition, 1993.

[HNW96] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential equations II : Stiff problems.* Springer-Verlag, second revised edition, 1996.

[Lam91] J. D. Lambert. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem.* John Wiley and Sons, 1991.

[Sha94] L. F. Shampine. *Numerical solution of ordinary differential equations.* Chapman and Hall, New York, 1994.