2 Runge–Kutta methods

2.1 The family of Runge–Kutta methods

In this section, we will introduce a family of increasingly accurate, and time-efficient, methods called *Runge–Kutta* methods after two German scientists: a mathematician and physicist Carl Runge (1856–1927) and a mathematician Martin Kutta (1867–1944).

The Modified Euler and Midpoint methods of the previous section can be written in a form common to both of these methods:

$$\begin{aligned} &Y_{i+1} = Y_i + (a\,k_1 + b\,k_2);\\ &k_1 = hf(x_i, Y_i),\\ &k_2 = hf(x_i + \alpha h, Y_i + \beta k_1);\\ &a,\ b,\ \alpha,\ \beta \quad \text{are some constants.} \end{aligned} \tag{2.1}$$

Specifically, for the Modified Euler,

$$a = b = \frac{1}{2}, \qquad \alpha = \beta = 1; \tag{2.2}$$

and for the Midpoint method,

$$a = 0, \quad b = 1, \qquad \alpha = \beta = \frac{1}{2}. \tag{2.3}$$

In general, if we require that method (2.1) have the global error $O(h^2)$, we can repeat the calculations we carried out in Section 1.4 for the Modified Euler method and obtain the following *3 equations* for *4 unknown coefficients* a, b, α, β:

$$a + b = 1, \qquad \alpha b = \frac{1}{2}, \qquad \beta b = \frac{1}{2}\,. \tag{2.4}$$

Observations:

- Since there are fewer equations than unknowns in (2.4), then there are infinitely many finite-difference methods whose global error is $O(h^2)$.
- One can generalize form (2.1) and seek methods of higher order (i.e. with the global error of $O(h^k)$ with $k \geq 3$) as follows:

$$\begin{aligned} &Y_{i+1} = Y_i + (ak_1 + bk_2 + ck_3 + \dots);\\ &k_1 = hf(x_i, Y_i),\\ &k_2 = hf(x_i + \alpha_2 h, Y_i + \beta_{21} k_1),\\ &k_3 = hf(x_i + \alpha_3 h, Y_i + \beta_{31} k_1 + \beta_{32} k_2),\\ &\text{etc.} \end{aligned} \tag{2.5}$$

  This family of methods is called the *Runge–Kutta (RK) methods.*

For example, if one looks for 4th-order methods, one obtains 11 equations for 13 coefficients. Again, this says that there are infinitely many 4th-order methods. Historically, the most popular

such method has been

$$\begin{aligned} Y_{i+1} &= Y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right); \\ k_1 &= hf(x_i, Y_i), \\ k_2 &= hf\left(x_i + \frac{1}{2}h, Y_i + \frac{1}{2}k_1\right), \\ k_3 &= hf\left(x_i + \frac{1}{2}h, Y_i + \frac{1}{2}k_2\right), \\ k_4 &= hf\left(x_i + h, Y_i + k_3\right). \end{aligned} \tag{2.6}$$

We will refer to this as the *classical* Runge–Kutta (cRK) method.

The table below compares the time-efficiency of the cRK and Modified Euler methods and shows that the former method is much more efficient.

Method	Global error	# of function evaluations per step
cRK	$O(h^4)$	4
Modified Euler	$O(h^2)$	2

One of the reasons why the cRK method is so popular is that the number of function evaluations per step in it equals the order of the method. It is known that RK methods of order $n \geq 5$ require *more than* n function evaluations; i.e. they are less efficient than the cRK and other lower-order RK methods. For example, a 5th-order RK method would require a minimum of 6 function evaluations per step.

2.2 Adaptive methods: Controlling step size for given accuracy

In this subsection, we discuss an important question of how the error of the numerical solution can be controlled and/or kept within a prescribed bound. A more complete and thorough discussion of this issue can be found in a paper by L.F. Shampine, "Error estimation and control for ODEs," SIAM J. of Scientific Computing, **25**, 3–16 (2005). A preprint of this paper is available on the course website.

To begin, we emphasize two important points about error control algorithms.

1. These algorithms control the *local truncation error*, and <u>not</u> the *global error*, of the solution. Indeed, the only way to control the global error is to run the simulations more than once. For example, one can run a simulation with the step h and then repeat it with the step $h/2$ to verify that the difference between the two solutions is within a prescribed accuracy. Although this can be done occasionally (for example, when confirming a key result of one's paper), it is too time-expensive to do so routinely. Therefore, the error control algorithms make sure that the local error at each step is less than a given tolerance (which is in some way related to the prescribed global accuracy), and then just let the user hope that the global accuracy is met. Fortunately, this hope comes true in most cases; but see the aforementioned paper for possible problematic cases.

2. The goal of the error control is not only to control the error but also to optimize the step size used to obtain different portions of the solution. For example, if it is found that the solution changes very smoothly on a subinterval $I_{\rm smooth}$ of the computational interval, then the step size on $I_{\rm smooth}$ can be taken sufficiently large. On the contrary, if one detects that the solution changes rapidly on another interval, $I_{\rm rapid}$, then the step size there should be decreased.

Methods where both the solution and its error are evaluated at each step of the calculation are called *adaptive* methods. They are most useful in problems with abruptly (or rapidly) changing coefficients. One simple example of such a problem is the motion of a skydiver: the air resistance changes abruptly at the moment the parachute opens. This will be discussed in more detail in the homework.

To present the idea of the algorithm used by adaptive methods, assume for the moment that we know the exact solution y_i. Let $\varepsilon_{\rm glob}$ be the maximum desired global error and n be the order of the method. Then the *actual* local truncation error must be $O(h^{n+1})$, or $ch^{n+1} + O(h^{n+2})$ with some constant c. Since the maximum allowed *local* truncation error, $\varepsilon_{\rm loc}$, is not prescribed, it has to be postulated in some plausible manner. The common choice is to take $\varepsilon_{\rm loc} = h\varepsilon_{\rm glob}$. Then, the steps of the **algorithm of an adaptive method** are as follows.

1. At each x_i, compute the actual local truncation error $\epsilon_i = |y_i - Y_i|$ and compare it with $\varepsilon_{\rm loc}$. (The practical implementation of this step is described later.)

2a. If $\epsilon_i < \varepsilon_{\rm loc}$, then accept the solution, multiply the step size by $\kappa(\varepsilon_{\rm loc}/\epsilon_i)^{1/(n+1)}$, (where κ is some numerical coefficient less than 1), and proceed to the next step.

2b. If $\epsilon_i > \varepsilon_{\rm loc}$, then multiply the step size by $\kappa(\varepsilon_{\rm loc}/\epsilon_i)^{1/(n+1)}$, re-calculate the solution, and check the error. If the actual error is acceptable, proceed to the next step. If not, repeat this step again.

Note that with the above step size adjustment, the error at the next step is expected to be approximately

$$c\left(h\cdot\kappa\left(\frac{\varepsilon_{\rm loc}}{\epsilon_i}\right)^{1/(n+1)}\right)^{n+1} \;=\; \varepsilon_{\rm loc}\kappa^{n+1}\,\frac{ch^{n+1}}{\epsilon_i} \;\approx\; \varepsilon_{\rm loc}\kappa^{n+1}\,.$$

The coefficient $\kappa < 1$ (say, $\kappa = 0.9$) is included to avoid the situation where the computed error just slightly exceeds the allowed bound, which would be acceptable to a human, but the computer will have to recalculate the entire step, thereby wasting expensive function evaluations.

Now, in reality, the exact solution of the ODE is not known. Then one can use the following trick. Suppose the numerical method we use is of sufficiently high order (e.g., the order 4 of the cRK method is sufficiently high for all practical purposes). Then we can compute the solution Y_i^h with the step size h and *at each step* compare it with the solution $Y_i^{h/2}$, obtained with the step size being halved. For example, for the cRK method is of fourth order, and hence $Y_i^{h/2}$ should be closer to the exact solution than Y_i^h is by about $2^4 = 16$ times. Then one can declare $Y_i^{h/2}$ *to be* the exact solution, compute $\tilde{\epsilon}_i = |Y_i^{h/2} - Y_i^h|$, and use $\tilde{\epsilon}_i$ in place of the ϵ_i above.

However, this way is very inefficient. For example, for the cRK method, it would require 7 additional function evaluations per step (needed to advance $Y^{h/2}$ from x_i to x_{i+1}). Therefore, people have designed alternative approaches to control the error size. Below we briefly describe the ideas behind two such approaches.

Runge–Kutta–Fehlberg method[1]

Idea: Design a 5th-order method that would *share* some of the function evaluations with a 4th-order method. The solution $Y_i^{[5]}$, obtained using the 5th-order method, is expected to be much more accurate than the solution $Y_i^{[4]}$, obtained using the 4th-order method. Then we declare $\tilde{\epsilon}_i = |Y_i^{[5]} - Y_i^{[4]}|$ to be the numerical error and adjust the step size based on that error relative to the allowed tolerance.

Implementation:

$$
\begin{aligned}
Y_{i+1}^{[4]} &= Y_i + \left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right), \\
Y_{i+1}^{[5]} &= Y_i + \left(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6\right); \\
k_1 &= hf(x_i, Y_i), \\
k_2 &= hf\left(x_i + \frac{1}{4}h, Y_i + \frac{1}{4}k_1\right), \\
k_3 &= hf\left(x_i + \frac{3}{8}h, Y_i + \frac{3}{32}k_1 + \frac{9}{32}k_2\right), \\
k_4 &= hf\left(x_i + \frac{12}{13}h, Y_i + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\right), \\
k_5 &= hf\left(x_i + h, Y_i + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right), \\
k_6 &= hf\left(x_i + \frac{1}{2}h, Y_i - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right), \\
&\text{where } Y_i = Y_i^{[5]}.
\end{aligned}
\tag{2.7}
$$

Altogether, there are only 6 function evaluations per step, because the 4th- and 5th-order methods share 4 function evaluations.

Runge–Kutta–Merson method

Idea: For certain choices of the auxiliary functions k_1, k_2, etc., the local truncation error of, say, a 4th order RK method can be made equal to $C_5 h^5 y^{(5)}(x_i) + O(h^6)$ with some known coefficient C_5. (Note that this local truncation error is proportional to the $(n+1)$-st derivative of the solution, where n is the order of the method. We observed a similar situation earlier for the simple Euler method; see Eq. (1.3).) On the other hand, a certain linear combination of the k's can also be chosen to equal $C_5 h^5 y^{(5)}(x_i) + O(h^6)$ *for a certain class of functions* (namely, for linear functions: $f(x, y) = a(x) + b \cdot y$, where $b = \text{const}$). Thus, we can obtain both an approximate solution and an estimate for its error. We can then use that estimate to adjust the step size so as to always make the (estimate for the) local truncation error below a prescribed maximum value.

For example, if one computes the solution Y_i using the cRK method and then, in addition, evaluates

$$
k_5 = hf\left(x_i + \frac{3}{4}h, Y_i + \frac{1}{32}\left[5k_1 + 7k_2 + 13k_3 - k_4\right]\right), \tag{2.8}
$$

[1] It is interesting to note that while the cRK method was developed in early 1900's, its extension by Fehlberg was proposed only in 1970.

then it can be shown (with a great deal of algebra) that

$$\text{Local truncation error } \sim \frac{2}{3}h\left(-k_1+3k_2+3k_3+3k_4-8k_5\right)+O(h^6)\,. \tag{2.9}$$

Here the sign '$\sim$' is used instead of the '=' because the equality holds only for $f(x,y) = a(x)+b\cdot y$, where $b=\text{const}$. Thus, again, by evaluating function f just one extra time compared to the cRK method, one obtains both the numerical solution and a *crude* estimate for its error. Then this error estimate can be used as the actual error ϵ_i in the algorithm of the corresponding adaptive method.

Implementation: More popular than the method described by (2.8) and (2.9), however, is another method based on the same idea and called the *Runge–Kutta–Merson* method:

$$\begin{aligned}
&Y_{i+1} = Y_i + \frac{1}{6}\left(k_1+4k_4+k_5\right);\\
&k_1 = hf(x_i,Y_i),\\
&k_2 = hf\left(x_i+\frac{1}{3}h, Y_i+\frac{1}{3}k_1\right),\\
&k_3 = hf\left(x_i+\frac{1}{3}h, Y_i+\frac{1}{6}(k_1+k_2)\right),\\
&k_4 = hf\left(x_i+\frac{1}{2}h, Y_i+\frac{1}{8}(k_1+3k_3)\right),\\
&k_5 = hf\left(x_i+h, Y_i+\frac{1}{2}(k_1-3k_3+4k_4)\right),\\
&\text{Local truncation error } \sim \tfrac{1}{30}(2k_1-9k_3+8k_4-k_5)\,.
\end{aligned} \tag{2.10}$$

Once again, one should note that the last line above is only a crude estimate for the truncation error (valid only when $f(x,y)$ is a linear function of y). Indeed, if it had been valid for any $f(x,y)$, then we would have a contradiction with a statement found at the end of Sec. 2.1. (Which statement is that?)

To conclude this presentation of the adaptive RK methods, we must specify what solution is taken at x_{i+1}. For example, for the RK–Fehlberg method, we have the choice between setting Y_{i+1} to either $Y^{[4]}_{i+1}$ or $Y^{[5]}_{i+1}$. The common sense suggests setting $Y_{i+1} = Y^{[5]}_{i+1}$, because, after all, it is $Y^{[5]}_{i+1}$ that we have declared to be our "etalon" solution. This choice does work in most circumstances, although there are important exceptions (see the paper by L. Shampine). Thus, what the RK–Fehlberg method does is *compute a 5th-order-accurate solution while controlling the error of a less accurate 4-th-order solution related to it.*

2.3 Questions for self-assessment

1. List the 13 coefficients mentioned in the paragraph after Eq. (2.5). Do <u>not</u> write the 11 equations.

2. If the step size is reduced by a factor of 2, how much will the error of the cRK and the Modified Euler methods be reduced? Which of these methods is more accurate?

3. Suppose $f = f(x)$ (on the r.h.s. of the ODE); that is, f does not depend on y but only on x. What numerical integration method (studied in Calculus 2) does the cRK method reduce to? [*Hint*: Rewrite Eq. (2.6) for $f = f(x)$.]

4. List the 7 function evaluations mentioned in the paragraph before the title 'Runge–Kutta–Fehlberg method'.

5. Describe the idea behind the Runge–Kutta–Fehlberg method.

6. Describe the idea behind the Runge–Kutta–Merson method.

7. Which statement is meant in the paragraph following Eq. (2.10)?

8. One of the built-in ODE solvers in MATLAB is called `ode45` . What do you think the origin of this name is? Without reading the description of this solver under MATLAB's help browser, can you guess what order this method is?